

| CONTENTS                                          | PAGE |
|---------------------------------------------------|------|
| EXERCISE 2                                        | 39   |
| A. Printing Buffer Contents—The Print Command "p" | 40   |
| EXERCISE 3                                        | 41   |
| A. The Current Line "." or Dot                    | 41   |
| B. Deleting Lines—The Delete Command "d"          | 42   |
| EXERCISE 4                                        | 43   |
| A. Modifying Text—The Substitute Command "s"      | 43   |
| EXERCISE 5                                        | 45   |
| A. Context Searching "/...../"                    | 45   |
| EXERCISE 6                                        | 47   |
| A. Change and Insert Commands "c" and "i"         | 47   |
| EXERCISE 7                                        | 48   |
| A. Moving Text Around—The Move Command "m"        | 49   |
| THE GLOBAL COMMANDS                               | 49   |
| SPECIAL CHARACTERS                                | 50   |
| SUMMARY OF COMMANDS AND LINE NUMBERS              | 52   |
| 5. AN INTRODUCTION TO SHELL                       | 55   |
| INTRODUCTION                                      | 55   |
| SIMPLE COMMANDS                                   | 55   |
| A. Background Commands                            | 55   |
| B. Input/Output Redirection                       | 56   |
| C. Pipelines and Filters                          | 56   |
| D. File Name Generation                           | 57   |
| E. Quoting                                        | 58   |
| F. Prompting by the Shell                         | 58   |
| G. The Shell and Login                            | 58   |



## CONTENTS

|                                                  | PAGE |
|--------------------------------------------------|------|
| H. Summary . . . . .                             | 59   |
| SHELL PROCEDURES . . . . .                       | 59   |
| A. Control Flow—for . . . . .                    | 60   |
| B. Control Flow—case . . . . .                   | 61   |
| C. Here Documents . . . . .                      | 62   |
| D. Shell Variables . . . . .                     | 63   |
| E. Test Command . . . . .                        | 65   |
| F. Control Flow—while . . . . .                  | 65   |
| G. Control Flow—if . . . . .                     | 66   |
| H. Debugging Shell Procedures . . . . .          | 68   |
| I. The "man" Command . . . . .                   | 69   |
| KEYWORD PARAMETERS . . . . .                     | 70   |
| A. Parameter Transmission . . . . .              | 70   |
| B. Parameter Substitution . . . . .              | 70   |
| C. Command Substitution . . . . .                | 71   |
| D. Evaluation and Quoting . . . . .              | 72   |
| E. Error Handling . . . . .                      | 74   |
| F. Fault Handling . . . . .                      | 75   |
| G. Command Execution . . . . .                   | 77   |
| H. Invoking the Shell . . . . .                  | 78   |
| 6. REMOTE JOB ENTRY (RJE) USER'S GUIDE . . . . . | 81   |
| INTRODUCTION . . . . .                           | 81   |
| GENERAL . . . . .                                | 81   |
| BASIC RJE . . . . .                              | 81   |
| A. Submitting Jobs . . . . .                     | 81   |
| B. Job Messages . . . . .                        | 82   |

# User's Guide Basic Dokumentation UNIX System

On Page 84 in the first paragraph under heading SEND COMMAND, change UNIX commands to UNIX system commands.

On Page 85 in the first sentence, change UNIX login to UNIX system login.

On Page 86 in the second paragraph after heading MONITORING RJE, change UNIX administrator to UNIX system administrator.



This dokument was prepared with specific references to use of the UNIX system on a particular processer, the Western Electric 3B20S, which is not presently available except for internal use within the Bell System. However, the information contained herein is generally applicable to use of the UNIX system on various processors which are available in the general trade.

Trademarks:

|                    |                              |
|--------------------|------------------------------|
| MUNIX, CADMUS      | for PCS                      |
| UNIX               | for Bell Laboratories        |
| DEC, PDP, VAX      | for DEC                      |
| MASSBUS, UNIBUS    |                              |
| NOVA, ECLIPSE      | for Data General Corporation |
| KODAK, EKTAMATIC   | for Eastman Kodak Company    |
| Mohrflow, Mohrdry, | for Mohr Lino-Saw Comp.      |
| Mohrchem           |                              |
| TEKTRONIX          | for Tektronik, Inc.          |
| TELETYPE           | for Teletype Corporation     |
| TRENDATA 4000A°    | for Trendata Corporation     |
| Versatec           | for Versatec Corporation     |
| DIABLO             | for Xerox Corporation        |

Copyright 1984 by

PCS GmbH, Pfälzer-Wald-Strasse 36, D-8000 München 90, tel. (089) 67804-0

The information contained herein is the property of PCS and shall neither be reproduced in whole or in part without PCS's prior written approval nor be implied to grant any license to make, use or sell equipment manufactured herewith.

PCS reserves the right to make changes without notice in the specifications and materials contained herein and shall not be responsible for any damages (including consequential) caused by reliance on the materials presented.

**USER'S GUIDE  
BASIC DOCUMENTATION  
UNIX SYSTEM**

| CONTENTS                               | PAGE |
|----------------------------------------|------|
| 1. INTRODUCTION . . . . .              | 7    |
| GENERAL . . . . .                      | 7    |
| 2. PRIMER . . . . .                    | 9    |
| INTRODUCTION . . . . .                 | 9    |
| HUMAN INTERFACE . . . . .              | 9    |
| A. Concept of a Login . . . . .        | 9    |
| B. Logging In . . . . .                | 10   |
| C. Logging Off . . . . .               | 10   |
| D. Entering Commands . . . . .         | 11   |
| E. Stopping a Program . . . . .        | 13   |
| F. Mail . . . . .                      | 13   |
| G. Writing to Other Users . . . . .    | 13   |
| H. On-line Manual . . . . .            | 14   |
| 3. BASICS FOR BEGINNERS . . . . .      | 17   |
| DAY-TO-DAY USE . . . . .               | 17   |
| A. Creating Files—The Editor . . . . . | 17   |
| B. What files are out there? . . . . . | 17   |
| C. Printing Files . . . . .            | 18   |
| D. Moving Files Around . . . . .       | 19   |
| E. What's in a File Name . . . . .     | 20   |



## CONTENTS

|                                                      | PAGE |
|------------------------------------------------------|------|
| F. What's in a File Name, Continued . . . . .        | 22   |
| G. Using Files Instead of the Terminal . . . . .     | 24   |
| H. Pipes . . . . .                                   | 25   |
| I. The Shell . . . . .                               | 26   |
| DOCUMENT PREPARATION . . . . .                       | 27   |
| A. Formatting Packages . . . . .                     | 28   |
| B. Supporting Tools . . . . .                        | 29   |
| C. Hints for Preparing Documents . . . . .           | 30   |
| D. Programming . . . . .                             | 30   |
| E. Shell Programming . . . . .                       | 30   |
| F. Programming with Shell . . . . .                  | 31   |
| G. Programming in C . . . . .                        | 32   |
| H. Other Languages . . . . .                         | 32   |
| GLOSSARY . . . . .                                   | 33   |
| 4. TUTORIAL—TEXT EDITOR . . . . .                    | 35   |
| INTRODUCTION . . . . .                               | 35   |
| GENERAL . . . . .                                    | 35   |
| A. Disclaimer . . . . .                              | 35   |
| GETTING STARTED . . . . .                            | 35   |
| A. Creating Text—The Append Command "a" . . . . .    | 36   |
| B. Error Messages (?) . . . . .                      | 37   |
| C. Writing Text File—The Write Command "w" . . . . . | 37   |
| D. Leaving ed—The Quit Command "q" . . . . .         | 37   |
| EXERCISE 1 . . . . .                                 | 38   |
| A. Reading Text File—The Edit Command "e" . . . . .  | 38   |
| B. Reading Text File—The Read Command "r" . . . . .  | 39   |

| CONTENTS                                             | PAGE |
|------------------------------------------------------|------|
| C. Output File Retention . . . . .                   | 83   |
| D. Examining Output Files . . . . .                  | 83   |
| SEND COMMAND . . . . .                               | 84   |
| JOB STREAM . . . . .                                 | 84   |
| USER FIELD SPECIFICATION . . . . .                   | 84   |
| A. Options . . . . .                                 | 84   |
| B. RJE Message Level of Notification . . . . .       | 86   |
| MONITORING RJE . . . . .                             | 86   |
| 7. SOURCE CODE CONTROL SYSTEM USER'S GUIDE . . . . . | 89   |
| GENERAL . . . . .                                    | 89   |
| SCCS FOR BEGINNERS . . . . .                         | 89   |
| A. Terminology . . . . .                             | 90   |
| B. Creating an SCCS File via "admin" . . . . .       | 90   |
| C. Retrieving a File via "get" . . . . .             | 90   |
| D. Recording Changes via "delta" . . . . .           | 91   |
| E. Additional Information About "get" . . . . .      | 92   |
| F. The "help" Command . . . . .                      | 93   |
| DELTA NUMBERING . . . . .                            | 93   |
| SCCS COMMAND CONVENTIONS . . . . .                   | 95   |
| SCCS COMMANDS . . . . .                              | 96   |
| A. The "get" Command . . . . .                       | 97   |
| ID Keywords . . . . .                                | 98   |
| Retrieval of Different Versions . . . . .            | 98   |
| Retrieval With Intent to Make a Delta . . . . .      | 100  |
| Concurrent Edits of Different SIDs . . . . .         | 101  |
| Concurrent Edits of Same SID . . . . .               | 102  |



## CONTENTS

|                                                                   | PAGE |
|-------------------------------------------------------------------|------|
| Keyletters That Affect Output . . . . .                           | 102  |
| B. The "delta" Command . . . . .                                  | 104  |
| C. The "admin" Command . . . . .                                  | 105  |
| Creation of SCCS Files . . . . .                                  | 106  |
| Inserting Commentary for the Initial Delta . . . . .              | 106  |
| Initialization and Modification of SCCS File Parameters . . . . . | 107  |
| D. The "prs" Command . . . . .                                    | 108  |
| E. The "help" Command . . . . .                                   | 109  |
| F. The "rmdel" Command . . . . .                                  | 109  |
| G. The "cdc" Command . . . . .                                    | 110  |
| H. The "what" Command . . . . .                                   | 110  |
| I. The "scsdiff" Command . . . . .                                | 111  |
| J. The "comb" Command . . . . .                                   | 111  |
| K. The "val" Command . . . . .                                    | 112  |
| SCCS FILES . . . . .                                              | 112  |
| A. Protecting . . . . .                                           | 112  |
| B. Formatting . . . . .                                           | 113  |
| C. Auditing . . . . .                                             | 114  |
| AN SCCS INTERFACE PROGRAM . . . . .                               | 114  |
| A. General . . . . .                                              | 114  |
| B. Function . . . . .                                             | 115  |
| C. A Basic Program . . . . .                                      | 115  |
| D. Linking and Use . . . . .                                      | 115  |

## 1. INTRODUCTION

### GENERAL

This section of the UNIX System User's Guide covers the following topics:

- A description of the features in the UNIX operating system
- A general overview of the capabilities of the UNIX operating system
- Instructions on how to use the UNIX operating system.

Not all of the capabilities of the UNIX operating system are described or illustrated herein, but enough are described so that a new user can become familiar with the use of the UNIX operating system. Using the available information, users who have more interest than the novice can utilize the information herein to accomplish their tasks with some experimenting and self-teaching.

Throughout this volume, each reference of the form **name(1M)**, **name(7)**, or **name(8)** refers to entries in the UNIX System Administrator's Manual. All other references to entries of the form **name(N)**, where "N" is a number (1 through 6) possibly followed by a letter, refer to entry **name** in section N of the UNIX System User's Manual.



NOTES

## 2. PRIMER

### INTRODUCTION

This section of the UNIX System User's Guide provides the information that users will need to know to access the UNIX operating system. It is not intended to be a detailed description. Many of the subjects described are discussed in detail in other sections of this volume or the UNIX System User's Manual.

Throughout this section, each reference of the form **name(1M)**, **name(7)**, or **name(8)** refers to entries in the UNIX System Administrator's Manual. All other references to entries of the form **name(N)**, where "N" is a number (1 through 6) possibly followed by a letter, refer to entry **name** in section N of the UNIX System User's Manual.

In this primer, software programs that can be executed by users are referred to as **programs**. A program that is in some state of execution is referred to as a **process**. The request typed by the user is referred to as a **command** or "command line."

In this primer, the following graphic conventions are used in examples:

- RETURN**                      Indicates that the user should press the RETURN key on the terminal keyboard.
- DEL**                              Indicates that the user should press the key marked DEL, DELETE, or RUBOUT (whichever is appropriate for the terminal being used).

### HUMAN INTERFACE

#### A. Concept of a Login

The UNIX operating system is accessed by the use of a **login**. A login is used by the system to uniquely identify users. Before the user can access the system, the user must be assigned a login by the system administrator. Every login consists of the following components:

- login name
- user identification number (uid)
- group identification number (gid)
- password.

A **login name** is a unique string of letters (should be all lowercase) and/or numbers that identifies an individual to the system. The login name must begin with a letter. In many cases, a person's login name is their real first name, last name, initials, or nickname. Any string of letters and/or digits can be used as your login name, as long as it is **unique** (i.e., different from all other login names). Only the first eight characters of a login name are used by the system. Login names are assigned by the system administrator.

The **uid** of a login is a unique number assigned to each login by the system administrator. This number is used by the system to identify the owners of information stored on the system and the commands that users are executing.

The **gid** is a unique number assigned by the system administrator to each group. This number identifies **groups** of users that have something in common. For example, all logins used by people in the same department (or working on the same project) may have the same gid. The gid is important for security and accounting reasons. The impact of gid numbers on the user and the group that the user belongs to is described later.

The **password** is a string of letters, numbers, and/or punctuation that serves to control access to a login. The password for a login is the main security feature of the UNIX operating system. Usually, every login is assigned a password. When a user **logs in** to the system, the password (if any) assigned to the login being used



is requested. Access to the system is not permitted until the correct password is entered. The user can change a password as needed to ensure that others are not accessing the user's login (and consequently the user's data). Any string of letters, numbers, etc., can be used as a password as long as it is more than five characters in length and composed of uppercase letters, lowercase letters, numbers, or punctuation.

It is recommended that obvious strings such as the user's social security number, birth date, or other data that could be well known about the user not be used as passwords. If the password is something that is well known about the user, someone could gain access to the user's login with little effort. The more unusual your password, the more effective your security.

#### B. Logging In

In order to log in, the power to the terminal must be turned on and the appropriate switches set. Depending on the type of terminal and communication link, the user may need to press the return or break key a couple of times. This is to synchronize your terminal with the system. When communication is established, the system will prompt with:

login:

The user should type in his/her login name followed by a return. After the system digests your login name, it will prompt for your password with:

Password:

The user should then type his/her password followed by a return. The system does not echo your password on the terminal as you type it in. This is an extra security measure. If you entered your login name and password correctly, the system may print one or more "messages of the day". Following the messages, the system will prompt you with the primary prompt string, which is usually the \$ symbol. If a mistake is made while logging in or the system administrator has not set up the user's login on the system yet, the following error message is printed:

login incorrect

This error message is followed by the login: message. The user should attempt to login again.

The UNIX operating system has a hierarchy of *directories*. When the system administrator gave the user a login name, the administrator also created a "directory" for the user. This directory ordinarily is the same name as the user login name and is known as the *login* or *home* directory of the user. When the user logs in, the *home* directory becomes the current directory or working directory of the user. Any file created under the login name (assuming no other subdirectories have been created yet) is by default in the home directory. The user may, however, create one or more directories under the *home* directory. The user may then change to subdirectories by appropriate use of a "change directory" command. See `cd(1)` for details. Under a directory or a subdirectory, the user may create files as necessary. The user is the owner of the *home* directory and all subdirectories created under the *home* directory. As the owner, the user has full permission to create, alter, and remove (destroy) all files and subdirectories of the *home* directory. To change from one directory to another, the command `cd` is used.

#### C. Logging Off

After completing your work, it is best to log off the system. Before logging off, you should have received the prompt string "\$" from the system. That is, all your commands have been completed and the system is ready for another command.

If you are using a "hard-wired" terminal, logging off is accomplished by typing an American Standard Code for Information Interchange (ASCII) End Of Text (EOT) character. On most terminals, the EOT character is



generated by holding down the "CONTROL" key and pressing the lowercase "d" key once. This is also referred to as a **CONTROL-d**. Another way to log off a hard-wired terminal is by simply turning the power to the terminal off.

For terminals connected via a phone line, the previously mentioned methods will also work, or instead, you can just hang up the phone.

Regardless of the type of terminal, the power to it should be turned off when the terminal is no longer needed. For terminals that use phone lines, depress the talk button.

#### D. Entering Commands

The UNIX operating system **shell** (command interpreter) serves as the interface between the user and the system. The **shell** accepts requests from the user in the form of a **command line** and invokes the appropriate program to fulfill the request. The **shell** prompts (i.e., notifies) the user when it is ready to accept another request. The prompt of the UNIX operating system **shell** is the primary prompt string which is by default "\$" (a dollar sign followed by a space).

#### Command Line Syntax

Commands or requests to the **shell** are usually in the form of a single line, that is, a string of one or more words followed by a return. This single line request entered following the prompt is referred to as a "command line". The command line is divided into two major parts—the program name and arguments.

The first word of the command line is the name of the program to be executed. This is referred to as the **command**. All subsequent words are *arguments* to the command. Arguments are used to provide information required by the requested program.

Spaces and tabs serve as the delimiters for words on the command line. That is, all characters on the command line up to the first space or tab is the command. All characters between the first space (or tab) and the second space (or tab) is the first argument, etc. Thus, the syntax for the command line is:

```
command argument argument argument ... ..(RETURN)
```

When spaces or tabs are needed within a single argument, that argument is enclosed by **double** quote marks. For example, to execute a program that requires two arguments such as john l and doe. The first argument should be john and the initial l, that is, "john l". The second argument should be doe. The required command line in this case would be:

```
command "john l" doe(RETURN)
```

#### Correction and Deletion

All users are likely to make mistakes, especially when typing. The UNIX operating system provides two features to correct command lines. These features are only effective for the current line (i.e., you have not ended the line with a return yet).

The first correction feature is the erase character (by default, #), and the second correction feature is the kill character (by default, @). The erase character erases the character preceding it. For example, a command line entered as

```
caf#t the fik#le (RETURN)
```



actually is "cat the file". The first # erases the first f and the second # erases the k. The erase character can be used to erase a series of characters such as in

```
this####the cat had kittens (RETURN)
```

which results in "the cat had kittens". The entire word "this" is erased by the series of # characters following it. The first # erases the s, the second # erases the i, the third # erases the h, and the fourth # erases the t. If you miscount the number of erase characters you need as in

```
this ###the cat had kittens (RETURN)
```

the result would be "ththe cat had kittens". The three erase characters erase the space, the s, and the i preceding them.

If the user needs to enter a # in the command line for some reason, preceding the # with the backslash character (\) will turn off the "erase last character" meaning of the #. For example, a command line entered as

```
thsi##is is the \#7# 7 cat (RETURN)
```

is actually "this is the # 7 cat".

The second correction feature is the kill character. The kill character deletes the entire current line. For example, the user enters the command line

```
command#####omma#####mmad argm##gmu##ment
```

when the user was trying to enter "command argument". This command line is so full of mistakes and corrections it is hard to determine if it is right. It would be best to delete the entire line and start over. The user can delete the line by ending it with an @ instead of a return. For example in this sequence

```
kat###catteh##he file##### the flie##e@
cat the file (RETURN)
```

the first line is deleted by the @ character. It is much easier to delete it and reenter it (as in the second line of the example).

If the @ character is needed in a line, the backslash character (\) should precede it. For example, entering the line

```
The kill character is a \@ (RETURN)
```

results in "The kill character is a @".

### ***Strange Terminal Behavior***

Sometimes you can get into a state where your terminal acts strangely. For example, each letter may be typed twice (terminal may be in the half-duplex mode) or the RETURN may not cause a line feed or a return to the left margin. The user can often change this by logging out and logging back in. If logging back in fails to correct the problem, the description of the `stty(1)` command can be read to determine the appropriate action to take. To get intelligent treatment of tab characters (which are much used in the UNIX operating system) if your terminal does not have tabs, type the command

```
stty -tabs
```

and the system will convert each tab into sufficient blanks to space to the next 8-character field. If your terminal does have hardware tabs, the command `tabs(1)` will set the stops correctly for you.



### *Read-ahead*

The UNIX operating system has full read-ahead, which means that the user can type as fast as desired, whenever the user wants, even when some command is typing at the user. If typing is done during output, the input characters will appear intermixed with the output characters, but they will be stored away and interpreted in the correct order. So the user can type several commands one after another without waiting for the first to finish or even begin.

#### **E. Stopping a Program**

Most programs can be stopped by typing the character "DEL" (perhaps called "delete" or "rubout" on your terminal). The "interrupt" or "break" key found on most terminals can also be used. In a few programs, like the text editor, "DEL" stops whatever the program is doing but leaves you in that program. Hanging up the phone will stop most programs.

#### **F. Mail**

After logging in, the user may sometimes get the following message:

You have mail.

The UNIX operating system provides a postal system so you can communicate with other users of the system. To read your mail, type the following command:

mail

Your mail will be printed, one message at a time, most recent message first. After each message, mail(1) waits for you to say what to do with it. The two basic responses are `d`, which deletes the message, and `RETURN`, which does not (so it will still be there the next time you read your mailbox). Other responses are described in the UNIX System User's Manual.

How is mail sent to someone else? Assume that "jones" is someone's login name which is recognized by `login(1)`. The easiest way to send mail to "jones" is as follows:

mail jones  
*now type in the text of the letter  
on as many lines as you like...*  
*After the last line of the letter*  
*type the character "CONTROL-d",*  
*that is, hold down "CONTROL" and type*  
*a letter "d".*

The "CONTROL-d" sequence, often called End-Of-File (EOF), is used throughout the system to mark the end of input from a terminal.

For practice, send mail to yourself. (This is not as strange as it might sound—mail to oneself is a handy reminder mechanism.)

There are other ways to send mail—you can send a previously prepared letter, and you can mail to a number of people all at once. For more details, see `mail(1)`.

#### **G. Writing to Other Users**

At some point, out of the blue will come a message like

Message from jones tty07...



accompanied by a startling beep. It means that Jones (jones) wants to talk to you; but unless you take explicit action, you will not be able to talk back. To respond, type the following command:

```
write jones
```

This establishes a 2-way communication path. Now whatever jones types on his terminal will appear on yours and vice versa. The path is slow, rather like talking to the moon. (If you are in the middle of something, you have to get to a state where you can type a command. Normally, whatever program you are running has to terminate or be terminated. If you are editing, you can escape temporarily from the editor—read the “Tutorial—Text Editor” section of this document.)

A protocol is needed to keep what you type from getting garbled up with what jones types. Typically it is like this:

Jones types “write smith” and waits.

Smith types “write jones” and waits.

Jones now types a message  
(as many lines as necessary).

When he is ready for a reply, he  
signals it by typing

(o)

which

stands for “over”.

Now Smith types a reply, also  
terminated by  
(o).

This cycle repeats until  
someone gets tired; he then  
signals his intent to quit with  
(oo)

for “over  
and out”.

To terminate  
the conversation, each side must  
type a “CONTROL-d” character alone  
at the beginning of a line. (“DELETE” also works.)  
When the other person types “CONTROL-d”,  
you will get the message  
EOF  
on your terminal.

If you write to someone who is not logged in or who does not want to be disturbed, you will be told. If the target is logged in but does not answer after a decent interval, simply type “CONTROL-d”.

#### H. On-line Manual

The UNIX System User's Manual is kept on-line. If you get stuck on something and can not find an expert to assist you, you can print on your terminal some manual section that might help. This is also useful for getting

the most up-to-date information on a command. To print a manual section, type "man command-name". Thus to read up on the `who(1)` command, type

`man who`

and, of course,

`man man`

tells all about the `man(1)` command.



USER'S GUIDE

ISSUE 1

6/82

*NOTES*

### 3. BASICS FOR BEGINNERS

#### DAY-TO-DAY USE

##### A. Creating Files—The Editor

If the users have to type a paper, a letter, or a program, how do they get the information stored in the machine? Most of these tasks are done with the UNIX operating system "text editor". See `ed(1)` for more details. Since the text editor is thoroughly documented in `ed(1)` and explained in the "Tutorial—Text Editor" section of this volume, no description is provided here on how to use it.

Throughout this section, each reference of the form `name(1M)`, `name(7)`, or `name(8)` refers to entries in the UNIX System Administrator's Manual. All other references to entries of the form `name(N)`, where "N" is a number (1 through 6) possibly followed by a letter, refer to entry `name` in section N of the UNIX System User's Manual.

A file is just a collection of information stored in the machine, this is a simplistic but adequate definition. The following text will describe how to make some *files*. To create a file called *junk* with text in it, do the following:

```
ed junk  (invokes the text editor)
a        (command to "ed" to add text)
now type in
whatever text you want ...
.        (signals the end of adding text)
```

The "." that signals the end of adding text must be at the beginning of a line by itself. Do not forget it, for until it is typed, no other `ed` commands will be recognized—everything you type will be treated as text to be added. Also note that no system prompt appears while you are appending, inserting, or changing text while in the text editor.

At this point the user can do various editing operations on the text which was typed in, such as correcting spelling mistakes, rearranging paragraphs, etc. Finally, the user must write the information typed into a file with the editor command:

w

The `ed` will respond with the number of characters it wrote into the file *junk*.

Nothing is stored permanently in the *junk* file until the `w` command is used. If the user is editing a file and hangs up before using the `w` command, the changes are not stored in the working file. The data in this case is saved in a file called *ed.hup* which the user can continue working with at the next editing session. But after `w` the information is there permanently. The user can reaccess it any time by typing the following:

ed junk

Type a `q` command to quit the editor. (If you try to quit without writing, the text editor will print a "?" to remind you. A second `q` gets the user out of the text editor regardless.) Now create a second file called *temp* in the same manner. You should now have two files, *junk* and *temp*.

##### B. What files are out there?

The `ls(1)` command lists the names (not contents) of any of the files that the UNIX operating system knows about. If you type

ls



the response will be

```
junk
temp
```

which are indeed the two files just created. The names are sorted into alphabetical order automatically, but other variations are possible. For example, the command

```
ls -t
```

causes the files to be listed in the order in which they were last changed, most recent first. The `-l` option gives a "long" listing and is used as follows:

```
ls -l
```

to produce something like

```
-rw-rw-rw- 1 bwk bsk 41 Jul 22 02:56 junk
-rw-rw-rw- 1 bwk bsk 78 Jul 22 12:57 temp
```

The date and time is the date and time of the last change to the file. The 41 and 78 are the number of characters (which should agree with the numbers you got from `ed`). The "bwk" is the owner of the file, i.e., the person who created it. The "bsk" identifies the group associated with "bwk". The "-rw-rw-rw-" determines who has permission to read, write, or execute the file. In this case the owner, group, and others all have permission to read (r) and write (w). Note that there is no permission for anyone to execute (x). The first character in "-rw-rw-rw-" is a "-" which indicates this is a file of data. A "d" in the first character would indicate a directory. The remaining nine characters are divided into three sets of permissions. Each set consists of three characters. The three sets correspond to the permissions of the owner, group, and all other users.

Options can be combined: `ls -lt` gives the same thing as `ls -l` but sorted into time order. The user can also name the files interested in, and `ls` will list the information about them only. More details can be found in `ls(1)`.

The use of optional arguments that begin with a minus sign (like `-t` and `-lt`) is a common convention for UNIX programs. In general, if a program accepts such optional arguments, they precede any file name arguments. It is also vital that you separate the various arguments with spaces: `ls-l` is not the same as `ls -l` since the command `ls` must be separated from its argument `-l` by a space. Try using the command both ways and observe the results.

### C. Printing Files

Now that you have created a file of text, how can the file be printed so people can look at it? There are several ways to print a file. One simple way to obtain a print is to use the editor, since printing is often done just before making changes anyway. The editor is used to print as follows:

```
ed junk
1,$p
```

The `ed` will reply with the count of the characters in *junk* and then print all the lines in the file. The user can also be selective about the parts of a file to be printed as follows:

```
ed junk
20,35p
```



There are times when it is not feasible to use the editor for printing. For example, there is a limit on how big a file `ed` can handle (several thousand lines). Secondly, it will only print one file at a time, and sometimes you want to print several, one after the other. So here are a couple of alternatives.

The simplest of all the printing programs is `cat(1)`. The `cat` simply prints on the terminal the contents of all the files named and in the order listed. Thus the files are concatenated and printed. For example:

```
cat junk
```

prints one file, and

```
cat junk temp
```

prints two files. The files are simply concatenated onto the terminal.

The `pr(1)` command produces formatted printouts of files. As with `cat`, `pr` prints all the files named in a list. The difference is that it produces headings with date, time, page number, and file name at the top of each page, and extra lines to skip over the fold in the paper. Thus,

```
pr junk temp
```

will print *junk* neatly, then skip to the top of a new page and print *temp* neatly.

The `pr` can also produce multicolumn output. Inputting

```
pr -3 junk
```

prints *junk* in 3-column format. You can use any reasonable number in place of "3" and `pr` will do its best. The `pr` command has other capabilities also. See `pr(1)` for more information.

It should be noted that `pr` is not a formatting program in the sense of shuffling lines around and justifying margins. The true formatters are `nroff` and `troff`, which we will get to in the section on document preparation.

There are also programs that print files on a hard copy printer. See `lp(1)` for more information.

#### D. Moving Files Around

The user is ready for bigger things after gaining experience in creating and printing files. For example, the user can move a file from one place to another (which amounts to giving it a new file name), like this:

```
mv junk precious
```

This means that what used to be named *junk* is now named *precious*. An `ls(1)` command would now result in the following:

```
precious
temp
```

The contents of *junk* are now in *precious*. Notice that the *junk* file no longer exists. Beware that if you move a file to another one that already exists, the already existing file contents are lost forever.

If you want to make a copy of a file (i.e., to have two versions of something), use the `cp(1)` command as follows:

```
cp precious templ
```



This makes a duplicate copy of *precious* in *templ*.

When you are finished creating and moving files, the files can be removed from the file system by the `rm(1)` command. The command is used as follows:

```
rm temp templ
```

This will remove both the *temp* and *templ* files.

The user will get a warning message if one of the named files is not there, but otherwise, `rm` like most UNIX system commands does its work silently. There is no prompting or response, and error messages are occasionally shortened. This terseness is sometimes disconcerting to newcomers, but experienced users find it desirable.

#### E. What's in a File Name

So far we have used file names without ever saying what is a legal name, so it is time for a couple of rules. First, file names are limited to 14 characters, which is enough to be descriptive. Second, although any character can be used in a file name, common sense dictates sticking to ones that are visible and avoiding characters that could be used with other meanings. We have already seen, for example, that in the `ls(1)` command, `ls -t` means to list in time order. So if a file existed whose name was `-t`, you would have a tough time listing it by name. Besides the minus sign, there are other characters which have special meaning. To avoid pitfalls, use only letters, numbers, and the period until you are familiar with the situation.

On to some more positive suggestions. Suppose you are typing a large document like a book. Logically, this divides into many small pieces, like chapters and perhaps sections. Physically, it must be divided too, for `ed` will not handle really big files. Thus the document should be typed as a number of files. One possible method is to have a separate file for each chapter as follows:

```
chap1
chap2
etc. ...
```

Another method is breaking each chapter into several files as follows:

```
chap1.1
chap1.2
chap1.3
...
chap2.1
chap2.2
...
```

It can now be determined at a glance where a particular file fits into the whole.

There are advantages to a systematic naming convention which are not obvious to the novice UNIX system user. To print the whole book, the user could enter the following:

```
pr chap1.a chap1.2 chap1.3 ...
```

Using the `pr(1)` command like this would be tiring and possibly lead to making mistakes. Fortunately, there is a shortcut. The user can enter:

```
pr chap*
```

The `*` means "anything at all", so this translates into "print all files whose names begin with *chap* listed in alphabetical order".



This shorthand notation is not a property of the `pr` command by the way. It is system-wide, a service of the program that interprets commands—the “`shell`”, `sh(1)`. The files in the book can be listed by using

```
ls chap*
```

which produces the following:

```
chap1.1
chap1.2
chap1.3
...
```

The `*` is not limited to the last position in a file name. The `*` can be used anywhere and can occur several times. Thus entering

```
rm *junk* *temp*
```

removes all files that contain *junk* or *temp* as any part of their name. As a special case, `*` by itself matches every file name, so

```
pr *
```

prints all your files (alphabetical order), and

```
rm *
```

removes all files. (Before using the `rm *` command, make sure all files are not needed!)

The `*` is not the only pattern-matching feature available. To print only chapters 1 through 4 and 9, use the following command:

```
pr chap[12349]*
```

The [...] means to match any of the characters inside the brackets. A range of consecutive letters or digits can be abbreviated as follows:

```
pr chap[1-49]*
```

Letters can also be used within brackets. The `[a-z]` pattern-matching feature matches any character in the range *a* through *z*.

The `?` pattern matches any single character, so

```
ls ?
```

lists all files which have single-character names, and

```
ls -l chap?.1
```

lists information about the first file of each chapter *chap1.1*, *chap2.1*, etc.

Of these niceties, `*` is certainly the most useful to become familiar with. The others are frills, but worth knowing.

If the special meaning of `*`, `?`, etc., needs to be turned off enclose the entire argument in single quotes as follows:

```
ls '?'
```



Some examples of this will be shown in the following paragraphs.

F. What's in a File Name, Continued

When the file called *junk* is first created, how did the system know that there was not another *junk* somewhere else, especially since the person in the next office could also be reading this tutorial? The answer is that generally each user has a private directory, which contains only the files that belong to that particular user. When you log in, you are "in" your directory. Unless the user takes special action when creating a new file, the new file is made in the directory that the user is currently in. This is most often your own directory, and thus the file is unrelated to any other file of the same name that might exist in another (someone else's) directory.

The set of all files is organized into a (usually big) tree with your files located several branches into the tree. It is possible for you to "walk" around this tree and find any file in the system by starting at the root of the tree and walking along the proper set of branches. Conversely, the user can start at their present location and walk toward the root.

Try the latter first. The basic tool is the command `pwd(1)` (print working directory) which prints the name of the directory the user is currently in.

Although the details will vary according to the system the user is on, the `pwd(1)` command will print something like:

```
/usr/your_name
```

This message indicates that the user is currently in the directory *your\_name*, which is in turn in the directory */usr*, which is in turn in the root directory called by convention just */*. (Even if it is not called */usr* on your system, the message will be something analogous. Make the corresponding changes and read on.)

If user now types

```
ls /usr/your_name
```

the results should be exactly the same list of file names as obtained from a plain `ls(1)`. With no arguments, `ls` lists the contents of the current directory. Given the name of a directory, it lists the contents of that directory.

Next, try using the following command:

```
ls /usr
```

This should print a long series of names, among which is your own login name *your\_name*. On many systems, *usr* is a directory that contains the directories of all the normal users of the system.

The next step is to try the following:

```
ls /
```

The response should be something like this (although again the details may be different):

```
bin
dev
etc
lib
```



```
tmp
usr
```

This is a collection of the basic directories of files that the system knows about; we are at the root of the tree.

If *junk* is still in your directory, enter the following:

```
cat /usr/your_name/junk
```

The name

```
/usr/your_name/junk
```

is called the *pathname* of the file that is normally thought of as *junk*. The *pathname* represents the full name of the path as followed from the root through the tree of directories to get to a particular file. It is a universal rule in the UNIX operating system that anywhere an ordinary file name can be used, the *pathname* can also be used.

This is not too exciting if all the files of interest are in your own directory; but if you work with someone else or on several projects concurrently, it becomes handy indeed. For example, your friends can print your book by entering the following:

```
pr /usr/your_name/chap*
```

Similarly, you can find out what files your neighbor has by entering:

```
ls /usr/neighbor
```

The "neighbor" just entered represents the login name of your neighbor. A copy of one of your neighbor's files can be made as follows:

```
cp /usr/neighbor/his_file your_file
```

If a file owner does not want someone else to have access to the owner's files, or vice versa, privacy can be arranged. Each file and directory has read-write-execute (rwx) permissions for the owner, a group, and everyone else, which can be set to control access. See *ls(1)* and *chmod(1)* for details. As a matter of observed fact, most users find openness of more benefit than privacy most of the time.

As a final experiment with pathnames, try the following:

```
ls /bin /usr/bin
```

Do some of the names look familiar? When a program is run, by typing its name after the prompt character, the system simply looks for a file of that name. It normally looks first in your directory (where it typically does not find it), then in */bin* and finally in */usr/bin*. There is nothing magic about commands like *cat(1)* or *ls(1)*, except that they have been collected into a couple of places to be easy to find and administer.

Two or more users can work regularly with common information in a friend's directory. This is accomplished by logging in as your friend. If you are already logged in as yourself and want to work in a friend's files, you could hang up and log in again as your friend. Another method is to simply change the current working directory as follows:

```
cd /usr/your_friend
```

Now when a file name is used in something like *cat(1)* or *pr(1)*, the command refers to the file in your friend's directory. Changing directories does not affect any permissions associated with a file. If you could not access



a file from your own directory, changing to another directory will not alter that fact. Of course, if you forget what directory you are in, type

```
pwd
```

to find out.

It is usually convenient to arrange your own files so that all the files related to one thing are in a directory separate from other projects. For example, when writing your book, the user might want to keep all the text in a directory called *book*. A directory can be made using the **mkdir(1)** command. The *book*-directory is made as follows:

```
mkdir book
```

The *book* directory can now be accessed to input chapters as follows:

```
cd book
```

If you logged in as yourself, the *pathname* of *book* is:

```
/usr/your_name/book
```

To remove the *book* directory, type:

```
rm book/*  
rmdir book  
or  
rm -r book
```

The **rm book/\*** command removes all files in the *book* directory. The **rmdir book** command is then used to remove the empty directory. The *book* directory must be empty before the **rmdir** command will work. The **rm -r book** command recursively deletes the entire contents of the *book* directory and then removes the *book* directory itself.

The user can go up one level in the tree of files by entering:

```
cd ..
```

The **".."** is the name of the parent of whatever directory you are currently in. For completeness, **"."** is an alternate name for the directory you are in.

#### G. Using Files Instead of the Terminal

Most of the commands used so far produce output on the terminal. Other commands, like the editor, take input from the terminal. It is universal in UNIX systems that the terminal can be replaced by a file for either or both of input and output. As one example,

```
ls
```

makes a list of files on your terminal. But if the user enters

```
ls >filelist
```

a list of your files will be placed in the file *filelist* (which will be created if it does not already exist or overwritten if it does). The symbol **>** means "put the output on the following file, rather than on the terminal". Nothing



is produced on the terminal. As another example, the user could combine several files into one by capturing the output of `cat` in a file:

```
cat f1 f2 f3 >temp
```

Another symbol, that operates very much like `>` does, is `>>`. The `>>` means "add to the end of". That is,

```
cat f1 f2 f3 >>temp
```

means to concatenate `f1`, `f2`, and `f3` to the end of whatever is already in `temp`, instead of overwriting the existing contents. As with `>`, if `temp` does not exist, it will be created.

In a similar way, the symbol `<` means to take the input for a program from the following file, instead of from the terminal. Thus, the user could make up a script of commonly used editing commands and put them into a file called `script`. The script could then be run on a file by entering:

```
ed file <script
```

Another example is using `ed` to prepare a letter in file `let`. The letter (file `let`) could then be sent to several people as follows:

```
mail adam eve mary joe <let
```

#### H. Pipes

One of the novel contributions of the UNIX operating system is the idea of a **pipe**. A pipe is simply a way to connect the output of one program to the input of another program, so the two run as a sequence of processes—a pipeline.

For example,

```
pr f g h
```

will print the files `f`, `g`, and `h`, beginning each on a new page. Instead of printing the files separately, the files can be printed together as follows:

```
cat f g h >temp  
pr <temp  
rm temp
```

This method is more work than necessary. To take the output of `cat` and connect it to the input of `pr`, use the following pipe:

```
cat f g h | pr
```

The vertical bar `|` means to take the output from `cat`, which would normally have gone to the terminal and put it into `pr` to be neatly formatted.

There are many other examples of pipes. For example,

```
ls | pr -3
```

prints a list of your files in three columns. The program `wc(1)` counts the number of lines, words, and characters in its input; and as seen earlier, the `who(1)` command prints a list of users currently logged on the system, one per access port. Thus

```
who | wc -l
```



tells how many people are logged on. And of course

```
ls | wc -l
```

counts your files.

Most programs that read from the terminal can read from a pipe instead. Most programs that write on the terminal can write on a pipe instead. There can be as many commands in a pipeline as desired.

Many UNIX operating system programs are written to take input from one or more files if file arguments are given. If no arguments are given, the programs will read from the terminal, and thus can be used in pipelines. One example using the `pr(1)` command to print files *a*, *b*, and *c* in three columns and in the order specified is as follows:

```
pr -3 a b c
```

But in

```
cat a b c | pr -3
```

the `pr` prints the information coming down the pipeline, still in three columns.

#### 1. The Shell

The mysterious “**shell**” mentioned previously is actually the `sh(1)` command. The shell is the program that interprets what is typed as commands and arguments. The shell also looks after translating \*, etc., into lists of file names, and <, >, and | into changes of input and output streams.

The shell has other capabilities too. For example, the user can run two programs with one command line by separating the commands with a semicolon. The shell recognizes the semicolon and breaks the line into two commands. Thus

```
date; who
```

does both commands before returning with a prompt character.

More than one program can run simultaneously if desired. This is beneficial when doing something time-consuming, like using the editor script. The act of running programs simultaneously prevents waiting around for the results before starting something else. An example follows:

```
ed file <script &
```

The ampersand at the end of a command line means “start this command running, then take further commands from the terminal immediately”, that is, do not wait for it to complete. Thus the script will begin, but the user can do something else at the same time. Of course, to keep the output from interfering with what you are doing on the terminal, it would be better to enter

```
ed file <script >script.out &
```

which saves the output lines in a file called *script.out*.

When a command is initiated with &, the system replies with a number called the process number. Programs running simultaneously can be terminated as follows:

```
kill process_number
```



The process number is used to identify the command to be stopped. If you forget the process number, the `ps(1)` command will list the process number for all programs you are running. (Entering `kill 0` will kill all your processes.) And if you are curious about other people, `ps -a` will provide information about all active programs that other users are currently running.

To start three commands that will execute in the order specified and in the background, enter the following:

```
(command_1; command_2; command_3) &
```

A background pipeline can be started as follows:

```
command_1 | command_2 &
```

Just as the editor or some similar program can get its input from a file instead of from the terminal, the shell can read a file to get commands. For instance, suppose the user wants to perform a sequence of actions after every log in such as:

- Set the tabs on the terminal
- Find out the date
- Find out who is on the system.

The three necessary commands to perform these actions [`tabs(1)`, `date(1)`, and `who(1)`] could be put in a file called *startup*. The *startup* file would then be run as follows:

```
sh startup
```

This instruction commands the machine to run the shell with the file *startup* as input. The effect is the same as typing the contents of *startup* on the terminal.

If this is to be a regular thing, the need to type `sh` every time can be eliminated by typing the following command only once:

```
chmod +x startup
```

To run the sequence of commands thereafter, the user only needs to enter:

```
startup
```

The `chmod(1)` command marks the file as being executable. The shell recognizes this and runs it as a sequence of commands.

If the user wants *startup* to run automatically for every log in, create a file in your login directory called *.profile* and place in it the line "startup". Upon logging in, the shell gains control and executes the commands found in the *.profile* file. We will get back to the shell in the section on programming.

#### DOCUMENT PREPARATION

UNIX operating systems are used extensively for document preparation. There are two major formatting programs, that is, programs that produce a text with justified right margins, automatic page numbering and



titling, automatic hyphenation, etc. The **nroff** program is designed to produce output on terminals and line-printers. The **troff** (pronounced "tee-roff") program was designed to drive a phototypesetter, which produces very high quality output on photographic paper. This document was formatted with **nroff**.

#### A. Formatting Packages

The basic idea of **nroff** (See **troff** for more information.) and **troff(1)** is that the text to be formatted contains within it "formatting commands" that indicate in detail how the formatted text is to look. For example, there may be commands that specify how long lines are, whether to use single or double spacing, and the running titles to use on each page.

Because **nroff** and **troff** are relatively hard to learn to use effectively, several "packages" of canned formatting requests are available to let you specify paragraphs, running titles, footnotes, multicolumn output, etc., with little effort and without having to learn **nroff** and **troff**. These packages take a modest effort to learn, but the rewards for using them are so great that it is time well spent.

This section provides a brief description of the "memorandum macros" package known as **mm(1)**. Formatting requests typically consist of a period and two uppercase letters, such as

.TL

which is used to introduce a title, or

.P

to begin a new paragraph.

The text of a typical document is entered so it looks something like this:

.TL

title

.AU "author information"

.MT "memorandum type"

.P

Enter text ---

---

.P

More text ---

---

.SG "signature"

The lines that begin with a period are the formatting macro requests. For example, **.P** calls for starting a new paragraph. The precise meaning of **.P** depends on the output device being used (typesetter or terminal, for instance) and the publication the document will appear in. For example, **-mm** normally assumes that a paragraph is preceded by a space—one line in **nroff** and one-half line in **troff** with the first word indented. These rules can be changed if desired, but they are changed by changing the interpretation of **.P**, not by retyping the document.

To actually produce a document in standard format using **-mm**, use the command

**troff -mm files ...**

for the typesetter, and

**nroff -mm files ...**



for a terminal. The `-mm` argument tells `troff` and `nroff` to use the manuscript package of formatting requests. There are several similar packages; check with a local expert to determine which ones are in common use on your machine.

## B. Supporting Tools

In addition to the basic formatters, there is a host of supporting programs that help with document preparation. The list in the next few paragraphs is far from complete, so browse through the UNIX System User's Manual and check with UNIX operating system users for other possibilities.

Both `eqn(1)` and `neqn` (See `eqn` for more information.) programs let you integrate mathematics into the text of a document in an easy-to-learn language that closely resembles the way you would speak it aloud. For example, the `eqn` input

```
sum from i=0 to n x sub i ~ = ~ pi over 2
```

produces the output

$$\sum_{i=0}^n x_i = \frac{\pi}{2}$$

The program `tbl(1)` provides an analogous service for preparing tables. The `tbl` program does all the computations necessary to align complicated columns with elements of varying widths.

The `spell(1)` program detects possible spelling mistakes in a document. The `spell` program compares the words in your document to a dictionary (stored in memory) and prints those words that are not in the dictionary. It knows enough about English spelling to detect plurals and the like, so it does a good job.

The `grep(1)` program looks through a set of files for lines that contain a particular text pattern (rather like the editor's context search does, but on a bunch of files). For example,

```
grep 'ing$' chap*
```

will find all lines that end with the letters *ing* in the files *chap\**. The "\$" indicated that the pattern to search for is at the end of the line, whereas a "^" indicates that the pattern to search for is at the beginning of a line. (It is almost always a good practice to put single quotes around the pattern to be searched for in case it contains characters like \* or \$ that have a special meaning to the shell.) The `grep` program is often used to locate the misspelled words detected by the `spell` program.

The `diff(1)` program prints a list of the differences between two files, so that two versions of something can automatically be compared. This is a vast improvement over proofreading by hand.

The `wc(1)` program counts the words, lines, and characters in a set of files. The `tr(1)` program translates characters into other characters. For example, `tr` will convert uppercase to lowercase and vice versa. This translates uppercase into lowercase:

```
tr [A-Z] [a-z] <input >output
```

The `sort(1)` program sorts files in a variety of ways while `cxref(1)` makes cross-references. The `ptx(1)` program makes a permuted index (keyword-in-context listing). The `sed(1)` program provides many of the editing



facilities of `ed` but can apply them to arbitrarily long inputs. The `awk(1)` program provides the ability to do both pattern matching and numeric computations and to conveniently process fields within lines. These programs are for more advanced users, and they are not limited to document preparation. Put them on your list of things to learn.

Most of these programs are either independently documented, in the supplemental package like `eqn(1)` and `tbl(1)` in the UNIX System Document Processing Guide, or the programs are sufficiently simple enough so that the description in the UNIX System User's Manual is an adequate explanation.

### C. Hints for Preparing Documents

Most documents go through several versions (always more than expected) before they are finally finished. Accordingly, you should do whatever possible to make the job of changing them easy.

First, when you do the purely mechanical operations of typing, type so that subsequent editing will be easy. Start each sentence on a newline. Make lines short, and break lines at natural places, such as after commas and semicolons, rather than randomly. Since most people change documents by rewriting phrases and adding, deleting, and rearranging sentences, these precautions simplify any editing needed later.

Keep the individual files of a document down to modest size, perhaps 10 to 15 thousand characters. Larger files edit more slowly. If a dumb mistake is made, it is better to clobber a small file than a big one. Split the files at natural boundaries in the document for the same reasons that you start each sentence on a newline.

The second aspect of making changes to documents easy is not to commit to the formatting details too early. One of the advantages of formatting packages is permitting format decisions to be delayed until the last possible moment. Indeed, until a document is printed, it is not even decided whether it will be typeset or printed out on a line printer.

As a rule of thumb, a document should be produced in terms of a set of requests or commands (macros) for all but the most trivial jobs. The macros used should then be defined either by using one of the existing macro packages (the recommended way) or by defining your own `nroff` and/or `troff` macros. As long as the text is entered in some systematic way, it can always be cleaned up and formatted by a judicious combination of editing commands and macro definitions.

### D. Programming

There will be no attempt made to teach any of the programming languages available but a few words of advice are in order. One of the reasons why the UNIX operating system is a productive programming environment is that there is already a rich set of tools available. Facilities like pipes, input/output redirection, and the capabilities of the shell often make it possible to do a job by pasting together programs that already exist instead of writing a program completely from scratch.

### E. Shell Programming

The pipe mechanism lets you fabricate quite complicated operations out of spare parts that already exist. For example, the first draft of the `spell` program was (roughly)

|                       |                                   |
|-----------------------|-----------------------------------|
| <code>cat ...</code>  | <i>collect the files</i>          |
| <code>  tr ...</code> | <i>put each word on a newline</i> |
| <code>  tr ...</code> | <i>delete punctuation, etc.</i>   |
| <code>  sort</code>   | <i>into dictionary order</i>      |
| <code>  uniq</code>   | <i>discard duplicates</i>         |
| <code>  comm</code>   | <i>print words in text</i>        |
|                       | <i>but not in dictionary</i>      |



More pieces have been added subsequently, but this goes a long way for such a small effort.

The editor can be made to do things that would normally require special programs on other systems. For example, to list the first and last lines of each of a set of files, such as a book, the user could laboriously type:

```
ed
e chap1.1
lp
$p
e chap1.2
lp
$p
etc.
```

The same job can be performed much more easily. One procedure is to type

```
ls chap* >temp
```

to get the list of file names into a file called *temp*. The *temp* file is then edited using global commands as follows:

```
1,$ s/^.*$/e & \
lp \
$p/
```

The results are written into the *script* file (1,\$ w script) and then the following command is entered:

```
ed <script
```

This will produce the same output as the laborious hand typing. Another method is using shell loops to repeat a set of commands over and over again for a set of arguments as illustrated below:

```
for i in chap*
do
    ed $i <script
done
```

This sets the shell variable *i* to each file name in turn, then does the command. This command can be entered at the terminal or put in a file for later execution. Before the file can be executed, it may be necessary to change the mode by entering the following:

```
chmod +x filename
```

#### F. Programming with Shell

An option often overlooked by new users is that the **shell** is itself a programming language, with variables, control flow **if-else**, **while**, **for**, **case**, subroutines, and interrupt handling. Since there are many building-block programs, the user can sometimes avoid writing a new program merely by piecing together some of the building blocks with shell command files.

We will not go into any details here; examples and rules can be found in the "Introduction to Shell" described later in this volume.



## G. Programming in C

The C language is a reasonable choice of a programming language when undertaking anything substantial. Everything in the UNIX operating system is based on C language. The system itself is written in C, as are most of the programs that run on the system. The C language is also an easy language to use once you get started. The C language is introduced and fully described in The C Programming Language by B. W. Kernighan and D. M. Ritchie (Prentice-Hall, 1978). Several sections of the manual describe the system interfaces, that is, how to do input/output and similar functions.

Most input and output in C is best handled with the standard input/output library, which provides a set of I/O functions that exist in compatible form on most machines that have C compilers. In general, it's wisest to confine the system interactions in a program to the facilities provided by this library. (Refer to Section 3 of the UNIX System User's Manual.)

The C programs that do not depend too much on the special features of the UNIX operating system (such as pipes) can be moved to other computers that have C compilers. The list of such machines grows daily; in addition to the PDP\*-11, it currently includes Honeywell 6000, IBM 370, Interdata 8/32, Data General Nova and Eclipse, HP 2100, Harris /7, VAX\*-11/780, Western Electric 3B20 and 3B5, and Zilog Z80. Calls to the standard I/O library will work on all of these machines.

There are a number of supporting programs that go with C. The `lint(1)` program checks C programs for potential portability problems and detects errors such as mismatched argument types and uninitialized variables.

For larger programs (anything whose source is on more than one file), the `make(1)` program allows you to specify the dependencies among the source files and the processing steps needed to make a new version. The program then checks the times that the pieces were last changed and does the minimal amount of recompiling to create a consistent updated version.

The debugger `sdb(1)` program is useful for digging through the dead bodies of C programs but is rather hard to learn to use effectively. The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.

The C compiler provides a limited statistical service, so a user can find where programs spend their time executing and what parts of a program are worth optimizing. Compile the programs with the `-p` option; after the test run, use `prof(1)` command to print a program execution profile. The command `time(1)` will give the gross run-time statistics of a program, but the times are not very accurate or reproducible.

## H. Other Languages

If Fortran must be used, there are two possibilities—Fortran 77 and `ratfor`. The user might consider `ratfor` which provides decent control structures and free-form input that characterize C, yet permits the writing of code that is also portable to other environments. Bear in mind that UNIX operating system Fortran tends to produce large and relatively slow-running programs. Furthermore, supporting software like `prof(1)`, etc., are all virtually useless with Fortran programs. If there is a Fortran 77 compiler on your system, it may be a viable alternative to `ratfor` and has the nontrivial advantage that it is compatible with the C language and related programs. (The `ratfor` processor and C tools can be used with Fortran 77 too.)

If your application requires translating a language into a set of actions or another language, the user is in effect building a compiler, though probably a small one. In that case, the `yacc(1)` compiler-compiler is recommended for use, which aids in developing a compiler quickly. The `lex(1)` lexical analyzer generator does the same job for the simpler languages that can be expressed as regular expressions. It can be used by itself or as a front

\* Trademark of Digital Equipment Corporation



end to recognize inputs for a **yacc-based** program. Both **yacc** and **lex** require some sophistication to use, but the initial effort of learning them can be repaid many times over in programs that are easy to change later.

#### GLOSSARY

**argument**—Words following the command on a command line that provide information necessary to execute a program.

**background**—A mode of program execution when the shell does not wait for the command to terminate before prompting for another command.

**command**—The first word of a command line. It is the name of an executable program.

**command line**—A request typed in by a user.

**current working directory**—The current point of reference for accessing data within the file system.

**kill character**—The character which is used to delete the current line, by default the kill character is @.

**directory**—A file system file type that is used to group and organize files and other directories.

**erase character**—The character which is used to delete the previous character on the current line. The default erase character is #.

**file**—A file system file type used to store information.

**foreground**—A mode of program execution when the shell waits for the command to terminate before prompting for another command.

**full pathname**—The pathname of a specific file starting from the **root** directory.

**group identification number (gid)**—A unique number assigned to one or more logins that is used to identify groups of related users.

**HOME**—Another name for the login directory.

**login**—A means by which a user can gain access to the UNIX operating system.

**login name**—A unique string of letters and numbers used to identify a login.

**mode**—In reference to files, protection.

**parent directory**—The directory immediately above another directory.

**partial pathname**—The pathname between the current working directory and a specific file.

**pathname**—A sequence of directory names separated by the / character and ending with the name of a file. The pathname defines the connection path between some directory and a file.

**process**—A program that is in some state of execution.

**program**—Software that can be executed by a user.

**shell**—A UNIX operating system program that handles the communication between the system and users. The shell accepts commands and causes the appropriate program to be executed.



***user identification number (uid)***—A unique number assigned to each login that is used to identify users and the owner of information stored on the system.



#### 4. TUTORIAL—TEXT EDITOR

##### INTRODUCTION

Almost all text input on the UNIX operating system is done with the standard text editor `ed(1)`. This is a tutorial guide to help beginners get started with text editing.

Although this guide does not cover everything about the UNIX operating system, it does discuss enough for most user's day-to-day needs. This includes printing, appending, changing, deleting, moving, and inserting entire lines of text; reading and writing files; context searching and line addressing; substituting; global changing; and using some special characters for easier editing.

Throughout this section, each reference of the form `name(1M)`, `name(7)`, or `name(8)` refers to entries in the UNIX System Administrator's Manual. All other references to entries of the form `name(N)`, where "N" is a number (1 through 6) possibly followed by a letter, refer to entry `name` in section N of the UNIX System User's Manual.

##### GENERAL

The `ed` program is a "text editor", that is, an interactive program for creating and modifying "text", using directions (commands) provided by a user at a terminal. The text is often a document like this one or perhaps data for a program.

This tutorial is meant to simplify learning `ed`. The recommended way to learn `ed` is to read this document, simultaneously using `ed` to follow the examples, then to read the description in Section 1 of the UNIX System User's Manual, all the while experimenting with `ed`. (Solicitation of advice from experienced UNIX operating system users is also useful.)

Do the exercises! The exercises illustrate techniques not completely discussed in the actual text. A summary at the end of this section summarizes the commands.

##### A. Disclaimer

This is a tutorial introduction and guide only. For this reason, no attempt is made to cover more than a part of the facilities that `ed` offers (although this fraction includes the most useful and frequently used facilities). Also, there is not enough space to explain the basic UNIX operating system procedures. It is assumed that the user knows how to log on to the UNIX operating system and has at least a vague understanding of a UNIX operating system file. For more on the UNIX operating system facilities, refer to the previous section of this volume titled, "Basics for Beginners".

The user must also know what character to type as the end-of-line character on the user's particular terminal. This character is the RETURN or newline character (key) on most terminals. Hereafter reference to the end-of-line character, whatever it is, will be referred to as RETURN.

##### GETTING STARTED

Assume that the user has logged in to a UNIX operating system and it has just printed the prompt character, usually a

\$

The easiest way to invoke `ed` is to type:

`ed` (followed by a RETURN)



You (the user) are now ready to go. The `ed` program is waiting to be told what to do.

#### A. Creating Text—The Append Command “a”

As your first problem, suppose some text is to be created starting from scratch. Perhaps the very first draft of a document or paper is to be entered; clearly, it will have to start somewhere and undergo modifications (editing) later. This part will describe how to enter some text to get a file of text started. How to make changes and corrections to the text is described later.

When `ed` is first invoked, it is rather like working with a blank piece of paper (the file)—there is no text or information present on the paper (in the file). The text must be supplied by the person using `ed`; it is usually done by typing in the text or by reading it into `ed` from a file. We will start by typing in some text and return shortly to how to read files.

First a bit of terminology. In `ed` jargon, the text being worked on is said to be “kept in a buffer.” Think of the buffer as a work space, if desired, or simply as the information that is to be edited. In effect the buffer is like the piece of paper on which we will write things, then change some of them, and finally file the whole thing away for another day.

The user tells `ed` what to do to the text by typing instructions called “commands.” Most commands consist of a single lowercase letter. Each command is typed on a separate line. (Sometimes the command is preceded by information about the line or lines of text to be affected—these will be described below.) The `ed` text editor makes no response to most commands—there is no prompting or response messages like “ready”. (This silence is preferred by experienced users, but sometimes is a hangup for new users.)

The first command is append, written as the letter

a

all by itself. It means “append (or add) text lines to the buffer, as I type them in.” Appending is rather like writing fresh material on a piece of paper. So to enter lines of text into the buffer, just type an

a

followed by a RETURN, and by the lines of text desired, like this:

a

```
Now is the time
for all good men
to come to the aid of their party.
```

The only way to stop appending is to type a line that contains only a period. The “.” is used to tell `ed` that the appending is finished. (Even experienced users forget to terminate appending with a “.” sometimes. If `ed` seems to be ignoring your entries, type an extra line with just the “.” on it. You may then find you have added some garbage lines to your text, which you will have to take out later.)

After the append command has been done, the buffer will contain the following three lines:

```
Now is the time
for all good men
to come to the aid of their party.
```

The `a` and the “.” are not there because they are not text.



To add more text to what already exists, just issue another `a` command and continue typing.

#### B. Error Messages (?)

If at any time the user makes an error in the commands typed into `ed`, the text editor will tell the user by typing the following:

?

This is about as cryptic as it can be, but with practice, the user can usually figure out the goof. The user can get a brief explanation of the error by typing

h

and maybe get some help.

#### C. Writing Text File—The Write Command "`w`"

It is likely that you will want to save your text for later use. To write out the contents of the buffer onto a file, use the write command

w

followed by the file name to write on. This will copy the buffer's contents onto the specified file (destroying any previous information on the file). To save (write) the text in a file named *junk*, for example, type:

w junk

Leave a space between `w` and the file name. The `ed` program will respond by printing the number of characters it wrote out. In this case, `ed` would respond with:

68

(Remember that blanks and the return character at the end of each line are included in the character count.) Writing a file just makes a copy of the text—the buffer's contents are not disturbed, so the user can go on adding lines to it. This is an important point. The `ed` program at all times works on a copy of a file, not the file itself. No change in the contents of a file takes place until you give a `w` command. (Writing out the text onto a file from time to time as it is being created is a good idea. If the system crashes or if the user makes some horrible mistake, all the text in the buffer will be lost but any text that was written onto a file is relatively safe.)

#### D. Leaving `ed`—The Quit Command "`q`"

To terminate a session with `ed`, first save your text by writing it onto a file using the `w` (write) command, and then type the `q` (quit) command:

q

The system will respond with the prompt character:

\$

At this point your buffer vanishes, with all its text, which is why the user would want to write before quitting. Actually `ed` will print the character

?



if the user tries to quit without writing. At this point, the user writes if desired; if not, another `q` will get you out regardless and will not save the text in the buffer.

## EXERCISES—TRY THEM!

### EXERCISE 1

Enter `ed` and create some text using the append command `a`

```
a
... text ...
```

Note that no system prompt appears while in the text editor. Do not forget to write the text into memory with the write command `w`. Write it into memory using the `w` command. Then leave `ed` with the `q` command and print the file to see that everything worked. To print a file, enter

```
pr filename
or
cat filename
```

in response to the prompt character (`$`). Try both.

#### A. Reading Text File—The Edit Command "`e`"

A common way to get text into the buffer is to read it from another file in the file system. This is what you do to edit text that you saved with the `w` command in a previous session. The edit command

```
e
```

retrieves the entire contents of a file into the buffer. So if the user had saved the three lines "Now is the time", etc., with a `w` command in an earlier session, the edit command

```
e junk
```

would place the entire contents of the file *junk* into the buffer and respond with a number

```
68
```

which is the number of characters in the file *junk*. If anything was already in the buffer, it is deleted first.

If the `e` command is used to read a file into the buffer, then the user does not need to use a file name after a subsequent `w` command; `ed` remembers the last file name used in an `e` command, and `w` will write on this file. Thus a good practice to follow is:

```
ed
e filename
[editing session]
.
w
q
```

This way, the user can simply enter `w` from time to time and be secure in the knowledge that if the user got the file name right at the beginning, the user is writing into the proper file each time. Note that after each edit command `e` or each write command `w` the number of characters is returned by `ed`.



The user can find out at any time what file name `ed` is remembering by typing the file command `f`. In this example, if you typed

```
f
```

`ed` would reply

```
junk
```

#### B. Reading Text File—The Read Command “`r`”

Sometimes you want to read a file into the buffer without destroying anything that is already in the buffer. This is done by the read command `r`. The command

```
r junk
```

will read the file *junk* into the buffer. The command appends the file specified to the end of whatever file is already in the buffer. So if you do a read after an edit command such as

```
e junk  
r junk
```

the buffer will contain two copies of the text (six lines) as follows:

```
Now is the time  
for all good men  
to come to the aid of their party.  
Now is the time  
for all good men  
to come to the aid of their party.
```

Like the `w` and `e` commands, `r` prints the number of characters read in after the reading operation is complete. Generally speaking, `r` is much less used than `e`.

The read command `r` may also be used to read a file external to the buffer into the file in the buffer. While in `ed` and at the current (or dot) line, enter the command

```
.r filename
```

and *filename* will be read into the file (already in the buffer) immediately after the current line. None of the file in the buffer is destroyed, rather the external file *filename* has been read into and been combined with the file already in the buffer. The file that was read remains in *filename* also. You only copied it. Notice the difference between “`r`” and “`.r`”.

#### EXERCISE 2

Experiment with the `e` command—try reading and printing various files. The user may get an error ?*name* where *name* is the name of a file. This means that the file does not exist. Some typical causes of getting an empty file are spelling the file name wrong or perhaps trying to read or write a particular file which your permissions will not allow. Try alternately reading and appending to see that they work similarly. Verify that

```
ed filename
```



is exactly equivalent to

ed  
e filename

What does

f filename

do?

#### A. Printing Buffer Contents—The Print Command "p"

To print or list the contents of the buffer (or parts of it) on the terminal, use the print command **p**. This is done as follows. Specify the line numbers where printing is to begin and end. These numbers have a comma between the beginning number and the ending number, i.e., "beginning line number, ending line number p". Thus to print the first ten lines of the contents of any buffer (i.e., lines 1 through 10), type:

1,10p (Prints lines 1 through 10)

The **ed** will respond by printing the specified starting line (1) through the specified ending line (10).

Suppose it is desirable to print all the lines in the buffer. You could use "1,30p" as above if it is known there were exactly 30 lines in the buffer. But in general, it is not known how many lines there are, so what can be used for the ending line number? The **ed** program provides a shorthand symbol for "line number of the last line in the buffer"—the dollar sign **\$**. To print all the lines in the buffer, use it this way:

1,\$p (Prints all lines in buffer)  
or  
,p (Prints all lines in buffer also)

This will print all the lines in the buffer (line 1 through the last line). The "1,\$p" can be abbreviated ",p". To stop the printing before the last line is printed, push the **DEL** key or the **DELETE** (or equivalent) key on the terminal. The **ed** program will respond

?

and wait for the next input command.

To print the last line of the buffer, use

,\$p

but **ed** lets you abbreviate this to

\$p

Any single line can be printed by typing the line number followed by a **p**. Thus

1p

produces the response

Now is the time

which is the first line of the buffer.



In fact, `ed` lets you abbreviate even further. You can print any single line by typing just the line number—no need to type the letter `p`. So by entering

`$`

`ed` will print the last line of the buffer. Entering a single line number will print that line only.

It is also possible to use `$` in combinations like

`$-5,$p`

which prints the last five lines of the buffer. This helps to determine the end of the contents of the buffer when more is to be entered.

### EXERCISE 3

Create some text using the `a` command and experiment with the `p` command. The user will find, for example, that line 0 or a line beyond the end (last line) of the buffer can not be printed. Also attempts to print a buffer in reverse order by entering

`3,lp`

will not work.

#### A. The Current Line "." or Dot

Suppose the buffer still contains the six lines of text (as in Exercise 1), and the following was entered

`1,3p`

and `ed` has printed the three lines. Try typing just

`p`

(no line numbers)

This will print

to come to the aid of their party.

which is the third line of the buffer. In fact it is the last (most recent) line that anything was done to. (The line just printed!) The `p` command can be repeated without line numbers, and it will continue to print line 3.

The reason is that `ed` maintains a record of the last line that anything was done to (in this case, line 3, which was just printed) so that it can be used instead of an explicit line number. This most recent line is referred to by the shorthand symbol

(Pronounced "dot").

Dot is a line number in the same way that `$` is. Dot means exactly "the current line", or loosely, "the line something was done to most recently." The dot can be used in several ways—one possibility is to enter:

`.,$p`

This will print all the lines from (including) the current line to the end (last line) of the buffer. In our example these are lines 3 through 6.



Some commands change the value of dot, while others do not. The print command **p** sets dot to the number of the last line printed; the last command will set both "." and \$ to the last line in the buffer (line 6).

Dot is most useful when used in combinations like this one:

.+1 (or equivalently, .+1p)

This means "print the next line" and is a handy way to step slowly through a buffer. The user can also enter

.-1 (or .-1p)

which means "print the line before the current line." This enables stepping through the buffer backwards if desired. Another useful one is something like

.-3,-1p

which prints the previous three lines.

Do not forget that all of these change the value of dot. The user can find out what dot is at any time by typing

.= (dot line number is ?)

The ed program will respond by printing the value (line number) of dot.

Let us summarize some things about the **p** command and dot. Essentially, **p** can be preceded by 0, 1, or 2 line numbers (for our example). If there is no line number given, it prints the "current line", the line that dot refers to. If there is one line number given (with or without the letter **p** it prints that line (and dot is set there); and if there are two line numbers (separated by a comma), it prints all the lines in that range (from the first number to the last number, and sets dot to the last line printed). If two line numbers are specified, the first can not be bigger than the second (refer to the beginning of "EXERCISE 3".)

Typing a single RETURN will cause printing of the next line—it is equivalent to:

.+1p

Try it. Typing a ^ is equivalent to typing the minus -. It can be used in multiples, as ^^^, which will move the current line or dot line backwards three lines from the current line. The "-" or the "^" can be considered equivalent to "-1p" since either moves the dot back one line.

#### B. Deleting Lines—The Delete Command "d"

Suppose three extra lines in the buffer are not needed. This is done by the delete command:

d

Except that **d** deletes lines instead of printing them, its action is similar to that of the print command **p**. The lines to be deleted are specified for **d** exactly as they are for **p** as follows:

starting line, ending line d

Thus the command

4,\$d

deletes lines 4 through the end. There are now three lines left, that can be checked by using:

1,\$p



And notice that \$ now is line 3! Dot is set to the next line after the last line deleted, unless the last line deleted is the last line in the buffer. In that case, dot is set to \$. The delete command **d** and the print command **p** may be used together thus

dp

which deletes the current line, prints the following line, and sets dot to the line printed.

#### EXERCISE 4

The user should experiment with **a**, **e**, **r**, **w**, **p**, and **d** until you become familiar with their use. While experimenting also use "dot", \$, and line numbers to understand their use.

When feeling adventurous, try using line numbers with **a**, **r**, and **c** as well. The user will find that **a** will append lines after the line number that you specify (rather than after dot); that **r** reads a file in after the line number you specify (not necessarily at the end of the buffer); and that **w** will write out exactly the lines specified, not necessarily the whole buffer. These variations are sometimes handy. For instance, a file can be inserted at the beginning of a buffer by entering:

Or filename

Lines can be entered at the beginning of the buffer by using:

0a  
...text...

Notice that ".w" is very different from,

w

#### A. Modifying Text—The Substitute Command "s"

We are now ready to try one of the most important of all commands—the substitute command

s

This is the command that is used to change individual words or letters within a line or group of lines. The substitute command is used for correcting spelling mistakes and typing errors.

Suppose that, because of a typing error, line 1 says

Now is th time

notice the **e** has been left off. The **s** command can be used to fix this as follows:

1s/th/the/

This says: in line 1, substitute for the characters th the characters the. To verify that it works, **ed** will not print the result automatically, enter

p



and get

Now is the time

which is what is desired. Notice that dot must have been set to the line where the substitution took place, since the **p** command printed that line. Dot is always set this way with the **s** command.

The general way to use the substitute command is

starting-line, ending-line s/change this/to this/

Whatever string of characters is between the first pair of slashes is replaced by whatever is between the second pair, in all the lines between starting-line and ending-line. Only the first occurrence on each line is changed however. If every occurrence is to be changed, see "EXERCISE 5". The rules for line numbers are the same as those for the print command **p** except that dot is set to the last line changed. (But there is a trap for the unwary: if no substitution took place, dot is not changed. This causes an error response ? as a warning.)

Thus the following can be entered

1,\$s/speling/spelling/

to correct the first spelling mistake (speling in this case) on each line in the text. (This is useful for people who are consistent misspellers!)

If no line numbers are given, the **s** command assumes we mean "make the substitution on line dot", so it changes things only on the current line. This leads to the very common sequence

s/something/something else/p

which makes some correction on the current line and then prints it (current line) to make sure it worked out right. If it did not, you can try again. (Notice that there is a **p** on the same line as the **s** command. With few exceptions, **p** can follow any command; no other multicommand lines are legal.)

It is also legal to say

s/.....//

which means change the first string of characters (.....) to nothing, i.e., remove them. This is useful for deleting extra words in a line or removing extra letters from words. For instance, if the buffer contained

Nowxx is the time

this can be corrected by entering

s/xx//p

to get

Now is the time

Notice that // (two adjacent slashes) means "no characters", not a blank. There is a difference! (See "Context Searching" under "EXERCISE 5" for another meaning of "//").



## EXERCISE 5

Experiment with the substitute command. See what happens if you substitute for some word on a line with several occurrences of that word. For example, enter

```
a
the other side of the coin
.
s/the/on the/p
```

which results in the following:

```
on the other side of the coin
```

A substitute command changes only the first occurrence of the first string. All occurrences can be changed by adding a **g** (for "global") command to the **s** command, like this:

```
s/.../.../gp
```

Try other characters instead of slashes to delimit the two sets of characters in the **s** command—anything should work except blanks or tabs.

If strange results are produced by inputting

```
^ . $ [ * \ &
```

read the part under "Special Characters" in this section.

#### A. Context Searching "/...../"

When the substitute command is mastered, move on to another highly important feature of **ed(1)**—context searching.

Suppose the original three lines of text in the buffer is as follows:

```
Now is the time
for all good men
to come to the aid of their party.
```

Suppose the word their is to be changed to the. How is the line that contains their located? With only three lines in the buffer, it is pretty easy to keep track of what line the word their is on. But if the buffer contained several hundred lines and you had been making changes, deleting and rearranging lines, etc., you would no longer really know what this line number would be. Context searching is simply a method of specifying the desired line, regardless of what its number is, by specifying some context (unique text) on it.

The way to say "search for a line that contains this particular string of characters" or "unique text" is to type:

```
/string of characters to find/
```

For example, the **ed** expression

```
/their/
```

is a context search which is sufficient to find the desired line—it will locate the next occurrence of the characters between slashes "their". It also sets dot to that line and prints that line for verification:

```
to come to the aid of their party.
```



"Next occurrence" means that `ed` starts looking for the string at line `+.1` and searches to the end of the buffer, then continues at line 1 and searches to line dot. That is, the search "wraps around" from `$` to 1. It scans all the lines in the buffer until it either finds the desired line or gets back to dot again. If the given string of characters can not be found in any line, `ed` types the error message

?

Otherwise, it prints the line it found.

The search for the desired line and the substitution can be done together, like this

```
/their/s/their/the/p
```

which will yield

```
to come to the aid of the party.
```

There were three parts to that last command: context search for the desired line, make the substitution, and print the line.

The expression `"/their/"` is a context search expression. In the simplest form, all context search expressions are like this—a string of characters surrounded by slashes. Context searches are interchangeable with line numbers, so they can be used by themselves to find and print a desired line or as line numbers for some other command, like `s`. They were used both ways in the examples above.

Suppose the buffer contains the three familiar lines

```
Now is the time
for all good men
to come to the aid of their party.
```

Then the `ed` line numbers

```
/Now/+1
/good/
/party/-1
```

are all context search expressions, and they all refer to the same line (line 2). To make a change in line 2, enter

```
/Now/+1s/good/bad/
or
/good/s/good/bad/
or
/party/-1s/good/bad/
```

The choice is dictated only by convenience. All three lines could be printed by entering

```
/Now/,/party/p
or
/Now/,/Now/+2p
```

or by any number of similar combinations. The first one of these might be better if you do not know how many lines are involved. (Of course, if there were only three lines in the buffer, a convenient method of printing would be

```
1,$p
```



but not if there were several hundred.)

The basic rule is: a context search expression is the same as a line number, so it can be used wherever a line number is needed.

#### EXERCISE 6

Experiment with context searching. Try a body of text with several occurrences of the same string of characters and scan through it using the same context search.

Try using context searches as line numbers for the substitute, print, and delete commands. They can also be used with `r`, `w`, and `a`.

Try context searching using `"?text?"` instead of `"/text/"`. This scans lines in the buffer in reverse order rather than normal (forward order). This is sometimes useful if you go too far while looking for some string of characters—it is an easy way to back up.

If funny results are obtained with any of the characters

`^ . $ [ * \ &`

read the part in this section on "Special Characters".

The `ed` program provides a short method for repeating a context search for the same string. For example, the `ed` line number

`/string/`

will find the next occurrence of "string". It often happens that this is not the desired line, so the search must be repeated. This can be done by typing merely:

`//`

This short method stands for "the most recently (last) used context search expression". It can also be used as the first string of the substitute command, as in

`/string1/s//string2/`

which will find the next occurrence of *string1* and replace it by *string2*. This can save a lot of typing. Similarly

`??`

means "scan backwards for the same expression."

#### A. Change and Insert Commands `"c"` and `"i"`

This section discusses the change command

`c`

which is used to change the current line or to replace the current line with a group of one or more lines, and the insert command

`i`



which is used for inserting a group of one or more lines immediately before the current line.

"Change", written as

c

is used to replace a number of lines with different lines, which are typed in at the terminal. For example, to change the first line (.+1) through the last line (\$) of a file to something else, type

.+1,\$c

...type the lines of text you want here...

The lines typed between the c command and the '.' (dot) command will take the place of the original lines between start line and end line. This is most useful in replacing a line or several lines which have errors.

If only one line is specified in the c command, then just that line is replaced. (You can type in as many replacement lines as you like.) Notice the use of '.' (dot) to end the input—this works just like the '.' (dot) in the a command and must appear by itself at the beginning of newline. If no line number is given, line dot is replaced. The value of dot is set to the last line you typed in.

"Insert" is similar to append—for instance

/string/i

...type the lines to be inserted here...

will insert the given text before the next line that contains "string". The text between i and the '.' (dot) is inserted before the specified line. If no line number is specified, the dot line is used. Dot is set to the last line inserted.

#### EXERCISE 7

"Change" is rather like a combination of delete followed by insert. Experiment to verify that

starting-line,ending-line d  
i  
...text...

is almost the same as

starting-line,ending-line c  
...text...

These are not precisely the same if the last line (\$) gets deleted. Check this out. What is dot?

Experiment with the append command a and the insert command i to see that they are similar but not the same. You will observe that

line-number a  
...text...



appends after the given line, while

```
line-number i
...text...
```

inserts before it. Observe that if no line number is given, **i** inserts before line dot, **a** appends after line dot, and **c** changes line dot.

#### A. Moving Text Around—The Move Command "m"

The move command **m** is used for cutting and pasting—it allows a group of lines to be moved from one place to another in the buffer. Suppose the first three lines of the buffer is to be placed at the end of the buffer instead of at the beginning. This could be performed by entering:

```
1,3w temp
$r temp
1,3d
```

(Do you see why?) This method will work, but it is a lot easier using the **m** command as follows:

```
1,3m$
```

The general case is:

```
starting-line,ending-line m after this line
```

Notice that there is a third line to be specified—the line after which the other lines are to be moved. Of course, the lines to be moved can be specified by context searches; if you had

```
First paragraph
...
end of first paragraph.
Second paragraph
...
end of second paragraph.
```

the two paragraphs could be reversed like this:

```
/Second/,/end of second/m/First/-1
```

Notice the "-1"—the moved text goes after the line mentioned. Dot gets set to the last line moved.

#### THE GLOBAL COMMANDS

The two global commands are **g** and **v**. The global command **g** is used to execute one or more **ed** commands on all those lines in the buffer that match some specified string. For example

```
g/peling/p
```

prints all lines that contain "peling". More usefully,

```
g/peling/s//pelling/gp
```



makes the substitution everywhere on the line, then prints each corrected line. Compare this to

```
1,$s/peling/pelling/gp
```

which only prints the last line substituted. Another subtle difference is that the **g** command does not give a ?—if “peling” is not found where the **s** command will.

There may be several commands including **a**, **c**, **i**, **r**, **w**, but not **g**; in that case, every line except the last must end with a backslash “\”.

```
g/xxx/-1s/abc/def/\
.+2s/ghi/jkl/\
.-2..p
```

makes changes in the lines before and after each line that contains “xxx”, then prints all three lines.

The **v** command is the same as **g** except that the commands are executed on every line that does not match the string following **v**. The following input

```
v/ /d
```

deletes every line that does not contain a blank.

## SPECIAL CHARACTERS

You may have noticed that things just do not work right when you used some characters like **.**, **\***, **\$**, and others in context searches and in the **s** command. The reason is rather complex, although the cure is simple. Basically, **ed** treats these characters as special, with special meanings. For instance, in a context search or the first string of the substitute command only,

```
/x.y/
```

means “a line with an **x**, any character, and a **y**”, not just “a line with an **x**, a period, and a **y**.”

The following is a complete list of the special characters that can cause trouble:

```
^ . $ [ * \ &
```

**Warning:** The backslash character “\” is special to “ed”. For safety’s sake, avoid it where possible.

If you have to use one of the special characters in a substitute command, you can turn off its magic meaning temporarily by preceding it with the backslash. Thus

```
s/\.\.\*/backslash dot star/
```

will change “\.\*” into “backslash dot star”.

Here is a brief synopsis of the other special characters. First, the circumflex “^” signifies the beginning of a line. Thus

```
/^string/
```



finds "string" only if it is at the beginning of a line. It will find

string

but not

the string...

The dollar-sign "\$" is just the opposite of the circumflex; it means the end of a line. The input

/string\$/

will only find an occurrence of "string" at the end of some line. This implies, of course, that

/^string\$/

will find only a line that contains just "string" and

/^.\$/

finds a line containing exactly one character.

The character ".", as we mentioned above, matches anything. For example, the input

/x.y/

matches any of the following:

x+y

x-y

x y

x.y

This is useful in conjunction with "\*" which is a repetition character. The "a\*" is a shorthand input for "any number of a's" therefore "." matches any number of anythings. For example input

s/.\*/stuff/

which changes an entire line, or

s/.\*,//

which deletes all characters in the line up to and including the last comma. (Since "." finds the longest possible match, this goes up to the last comma.)

The "[" is used with the "]" to form character classes; for example,

/[0123456789]/

matches any single digit—any one of the characters inside the braces will cause a match. This can be abbreviated to

[0-9]

Finally, the "&" is another shorthand character—it is used only on the right-hand part of a substitute command where it means "whatever was matched on the left-hand side". It is used to save typing. Suppose the current line contained

Now is the time



and you wanted to put parentheses around it. One tedious method is just to retype the line. Another method is to enter

```
s/^/(/
s/$/)/
```

using your knowledge of "^" and "\$". But the easiest way uses the "&" as follows:

```
s/.*/(&)/
```

This says "match the whole line and replace it by itself surrounded by parentheses." The "&" can be used several times in a line; consider using

```
s/.*/&? &!!/
```

to produce

```
Now is the time? Now is the time!!
```

You do not have to match the whole line, of course. If the buffer contains

```
the end of the world
```

you could type

```
/world/s//& is at hand/
```

to produce

```
the end of the world is at hand
```

Observe this expression carefully, for it illustrates how to take advantage of **ed** to save typing. The string `/world/` found the desired line; the shorthand `//` found the same word in the line; and the `&` saves you from typing it again.

The `&` is a special character only within the replacement text of a substitute command and has no special meaning elsewhere. You can turn off the special meaning of `&` by preceding it with a backslash `\`. Inputting

```
s/ampersand/\&/
```

will convert the word "ampersand" into the literal symbol `&` in the current (dot) line.

#### SUMMARY OF COMMANDS AND LINE NUMBERS

The general form of the **ed** text editor commands is the command **name**, perhaps preceded by one or two line numbers. In the case of the edit command **e**, the read command **r**, and the write command **w**, the command **name** is also followed by a *file name*. Normally only one command is allowed to be entered per line, but a print command **p** may follow any other command (except for the edit command **e**, the read command **r**, the write command **w**, and the quit command **q**).

a

Append adds lines to the buffer (at line dot, unless a different line is specified). Appending continues until a dot "." is typed at the beginning (first character) of a newline. Dot is set to the last line appended.



- c** Change the specified lines to the new text which follows. Entering newlines is terminated by a dot "." as with **a**. If no lines are specified, the current line (dot) is replaced. Dot is set to last line changed.
- d** Delete the lines specified. If none are specified, delete line dot. Dot is set to the first undeleted line, unless **\$** is specified in which case dot is set to the last line, **\$**.
- e** Edit new file. Any previous contents of the buffer are thrown away, so issue a write command **w** beforehand.
- f** Print the remembered *file* name. If a name follows **f**, the remembered name will be set to it.
- g** The global command **g/---/commands** will execute the commands on those lines that contain "---".
- i** Insert lines before the specified line or the current line (dot line) until a "." is typed at the beginning of a newline. Dot is set to last line inserted.
- m** Move lines specified to after the line named after **m**. Dot is set to the last line moved.
- p** Print specified lines. If none specified, print line dot. A single line number is equivalent to "line number". A single RETURN prints the next line, i.e., the dot plus one line, ".+1".
- q** The quit command exits from **ed**. It wipes out all text in buffer if you give it twice in a row without first giving a write command **w**.
- r** Read a file into buffer (at end unless specified elsewhere). Dot set to last line read. If **.r filename** is used, the filename is read into the buffer immediately after the dot line.
- s** The substitute command **s/string1/string2/** substitutes the characters "string1" into "string2" in the specified lines. If no lines are specified, make the substitution in line dot. Dot is set to last line in which a substitution took place; if no substitution took place, dot is not changed. The command **s** changes only the first occurrence of "string1" on a line; to change all occurrences on a line, type a **g** after the final slash.
- v** The exclude command **v/---/commands** executes commands only on those lines that do not contain "---".
- w** The write command writes out the buffer contents onto a file. Dot is not changed.
- .=** The **."** causes the printout of the current line number. The dot value prints the line number of the current line (dot line). The **"=** by itself prints the value of the last line in the file.
- !** The **"!** is a temporary escape command. The line **!** command-line causes "command-line" to be executed as a UNIX operating system command while you are in the text editor.
- /-----/** The context search command searches for next line which contains this string of characters "-----" and prints it. Dot is set to the line where string was found. Search starts at line **.=1** wraps around from the last line **"\$"** to line **"1"** and continues to dot (the current line), if necessary.
- ?-----?** Performs context search in reverse direction. Starts search at the previous line **"-1"**, scans to line 1, wraps around to the last line **"\$"**, and scans back to the current line (dot line) if necessary.



NOTES



## 5. AN INTRODUCTION TO SHELL

### INTRODUCTION

The **shell** is a command programming language that provides an interface to the UNIX operating system. Its features include control-flow primitives, parameter passing, variables, and string substitution. Constructs such as **while**, **if then else**, **case**, and **for** are available. Two-way communication is possible between the shell and commands. String-valued parameters, typically file names or flags, may be passed to a command. A return code is set by commands that may be used to determine control-flow, and the standard output from a command may be used as **shell** input.

The **shell** can modify the environment in which commands run. Input and output can be redirected to files, and processes that communicate through **pipes** can be invoked. Commands are found by searching directories in the file system in a sequence that can be defined by the user. Commands can be read either from the terminal or from a file which allows command procedures to be stored for later use.

The **shell** is both a command language and a programming language that provides an interface to the UNIX operating system. This volume describes, with examples, the UNIX operating system **shell**. The "Simple Commands" part of this section covers most of the everyday requirements of terminal users. Some familiarity with UNIX operating system is an advantage when reading this section; refer to the section "BASICS for Beginners". The "Shell Procedures" part of this section describes those features of the **shell** primarily intended for use within **shell** commands or procedures. These include the control-flow primitives and string-valued variables provided by the **shell**. A knowledge of a programming language would be helpful when reading this section. The last part, "Keyword Parameters", describes the more advanced features of the **shell**. See Table 5.A for a defined listing of grammar words used in this section.

Throughout this section, each reference of the form **name(1M)**, **name(7)**, or **name(8)** refers to entries in the UNIX System Administrator's Manual. All other references to entries of the form **name(N)**, where "N" is a number (1 through 6) possibly followed by a letter, refer to entry **name** in section N of the UNIX System User's Manual.

### SIMPLE COMMANDS

Simple commands consist of one or more words separated by blanks. The first word is the **name** of the command to be executed; any remaining words are passed as **arguments** to the command. For example,

who

is a command that prints the names of users logged in. The command

ls -l

prints a list of files in the current directory. The argument **-l** tells **ls(1)** to print status information, size, and the creation date for each file.

#### A. Background Commands

To execute a command, the **shell** normally creates a new process and waits for it to finish. A command may be run without waiting for it to finish. For example,

cc pgm.c &

calls the C compiler to compile the file **pgm.c**. The trailing "&" is an operator that instructs the **shell** not to wait for the command to finish. To help keep track of such a process, the **shell** reports its process number following its creation. A list of currently active processes may be obtained using the **ps(1)** command.



## B. Input/Output Redirection

Most commands produce output to the standard output that is initially connected to the terminal. This output may be directed to a file by the notation ">" thus:

```
ls -l >file
```

The notation >*file* is interpreted by the **shell** and is not passed as an argument to **ls(1)**. If *file* does not exist, the **shell** creates it; otherwise, the original contents of *file* are replaced with the output from **ls(1)**. Output may be appended to a file using the notation ">>" as follows:

```
ls -l >>file
```

In this case, *file* is also created if it does not already exist.

The standard input of a command may be taken from a file instead of the terminal by the notation "<" thus:

```
wc <file
```

The command **wc(1)** reads its standard input (in this case redirected from *file*) and prints the number of characters, words, and lines found. If only the number of lines is required, then

```
wc -l <file
```

can be used.

## C. Pipelines and Filters

The standard output of one command may be connected to the standard input of another by writing the "pipe" operator, indicated by **|**, between commands as in

```
ls -l | wc
```

Two or more commands connected in this way constitute a **pipeline**, and the overall effect is the same as

```
ls -l >file; wc <file
```

except that no *file* is used. Instead the two processes are connected by a pipe [see **pipe(2)**] and are run in parallel. Pipes are unidirectional, and synchronization is achieved by halting **wc(1)** when there is nothing to read and halting **ls(1)** when the pipe is full.

A **filter** is a command that reads its standard input, transforms it in some way, and prints the result as output. One such filter, **grep(1)** selects from its input those lines that contain some specified string. For example,

```
ls | grep old
```

prints those lines, if any, of the output from **ls** that contain the string "old". Another useful filter is **sort(1)**. For example,

```
who | sort
```

will print an alphabetically sorted list of logged in users.

A pipeline may consist of more than two commands, for example,

```
ls | grep old | wc -l
```



prints only the number of file names in the current directory containing the string "old".

#### D. File Name Generation

Many commands accept arguments which are file names. For example,

```
ls -l main.c
```

prints only information relating to the file *main.c*. The "`ls -l`" command alone prints the same information about all files in the current directory.

The **shell** provides a mechanism for generating a list of file names that match a pattern. For example,

```
ls -l *.c
```

generates as arguments to `ls(1)` all file names in the current directory that end in *.c*. The character "\*" is a pattern that will match any string including the null string. In general, patterns are specified as follows:

- \* Matches any string of characters including the null string.
- ? Matches any single character.
- [...] Matches any one of the characters enclosed. A pair of characters separated by a minus will match any character lexically between the pair.

For example,

```
[a-z]*
```

matches all names in the current directory beginning with one of the letters *a* through *z*. The input

```
/usr/fred/test/?
```

matches all names in the directory */usr/fred/test* that consist of a single character. If no file name is found that matches the pattern, then the pattern is passed, unchanged, as an argument.

This mechanism is useful both to save typing and to select names according to some pattern. It may also be used to find files. For example,

```
echo /usr/fred/*/core
```

finds and prints the names of all *core* files in subdirectories of */usr/fred*. [The `echo(1)` command is a standard UNIX operating system command that prints its arguments, separated by blanks.] This last feature can be expensive requiring a scan of all subdirectories of */usr/fred*.

There is one exception to the general rules given for patterns. The character "." at the start of a file name must be explicitly matched. The input

```
echo *
```

will therefore echo all file names in the current directory not beginning with ".". The input

```
echo .*
```



will echo all those file names that begin with ".". This avoids inadvertent matching of the names "." and ".." which mean "the current directory" and "the parent directory", respectively. [Notice that `ls(1)` suppresses information for the files "." and "..".]

#### E. Quoting

Characters that have a special meaning to the **shell**, such as

< > \* ? | &

are called **metacharacters**. A complete list of metacharacters is given in Table 5.B. Any character preceded by a \ is **quoted** and loses its special meaning, if any. The \ is elided so that

`echo \?`

will echo a single ?, and

`echo \\\`

will echo a single \. To allow long strings to be continued over more than one line, the sequence `\newline` (or `RETURN`) is ignored. The \ is convenient for quoting single characters. When more than one character needs quoting, the above mechanism is clumsy and error prone. A string of characters may be quoted by enclosing the string between single quotes. For example,

`echo xx'****'xx`

will echo

`xx****xx`

The quoted string may not contain a single quote but may contain newlines which are preserved. This quoting mechanism is the most simple and is recommended for casual use. A third quoting mechanism using double quotes is also available and prevents interpretation of some but not all metacharacters. Details of quoting are described under "D. Evaluation and Quoting" in part "Keyword Parameters".

#### F. Prompting by the Shell

When the **shell** is used from a terminal, it will issue a prompt to the terminal user indicating it is ready to read a command from the terminal. By default, this prompt is "\$". The prompt may be changed by entering,

`PS1=newprompt`

that sets the prompt to be the string "newprompt". If a newline is typed and further input is needed, the **shell** will issue the prompt ">". Sometimes this can be caused by mistyping a quote mark. If it is unexpected, then an interrupt (DEL) will return the **shell** to read another command. The other prompt (>) may be changed by entering:

`PS2=more`

#### G. The Shell and Login

Following the user's `login(1)`, the **shell** is called to read and execute commands typed at the terminal. If the user's login directory contains the file `.profile`, then it is assumed to contain commands and is read immediately by the **shell** before reading any commands from the terminal.



## H. Summary

```
ls
Print the names of files in the current directory.
ls >file
Put the output from ls into file.
ls | wc -l
Print the number of files in the current directory.
ls | grep old
Print those file names containing the string "old".
ls | grep old | wc -l
Print the number of files whose name contains
the string "old".
cc pgm.c &
Run cc in the background.
```

## SHELL PROCEDURES

The **shell** may be used to read and execute commands contained in a file. For example, the following call

```
sh file [ args ... ]
```

calls the **shell** to read commands from *file*. Such a file call is called a "command procedure" or "shell procedure". Arguments may be supplied with the call and are referred to in *file* using the positional parameters **\$1**, **\$2**, ... . For example, if the file *wg* contains

```
who | grep $1
```

then the call

```
sh wg fred
```

is equivalent to

```
who | grep fred
```

All UNIX operating system files have three independent attributes (often called "permissions"), **read**, **write**, and **execute** (rwx). The UNIX operating system command **chmod(1)** may be used to make a file executable. For example,

```
chmod +x wg
```

will ensure that the file *wg* has execute status (permission). Following this, the command

```
wg fred
```

is equivalent to the call

```
sh wg fred
```

This allows **shell** procedures and programs to be used interchangeably. In either case, a new process is created to execute the command.

As well as providing names for the positional parameters, the number of positional parameters in the call is available as **\$#**. The name of the file being executed is available as **\$0**.



A special **shell** parameter **\$\*** is used to substitute for all positional parameters except **\$0**. A typical use of this is to provide some default arguments, as in,

```
nroff -T450 -cm $*
```

which simply prepends some arguments to those already given.

#### A. Control Flow—**for**

A frequent use of **shell** procedures is to loop through the arguments (**\$1**, **\$2**, ...) executing commands once for each argument. An example of such a procedure is *tel* that searches the file */usr/lib/telnet* that contains lines of the form

```
...
fred mh0123
bert mh0789
...
```

The text of *tel* is

```
for i
do
grep $i /usr/lib/telnet; done
```

The command

```
tel fred
```

prints those lines in */usr/lib/telnet* that contain the string "fred".

The command

```
tel fred bert
```

prints those lines containing "fred" followed by those for "bert".

The **for** loop notation is recognized by the **shell** and has the general form

```
for name in w1 w2
do
    command-list
done
```

A **command-list** is a sequence of one or more simple commands separated or terminated by a newline or a semicolon. Furthermore, reserved words like **do** and **done** are only recognized following a newline or semicolon. A *name* is a **shell** variable that is set to the words *w1 w2 ...* in turn each time the **command-list** following **do** is executed. If "in *w1 w2 ...*" is omitted, then the loop is executed once for each positional parameter; that is, in **\$\*** is assumed.

Another example of the use of the **for** loop is the **create** command whose text is

```
for i do >$i; done
```

The command

```
create alpha beta
```



ensures that two empty files *alpha* and *beta* exist and are empty. The notation *>file* may be used on its own to create or clear the contents of a file. Notice also that a semicolon (or newline) is required before **done**.

#### B. Control Flow—case

A multiple way (choice) branch is provided for by the **case** notation. For example,

```
case $# in
  1) cat >>$1 ;;
  2) cat >>$2 <$1 ;;
  *) echo\*'usage: append [ from ] to' ;;
esac
```

is an append command. (Note the use of semicolons to delimit the cases.) When called with one argument as in

```
append file
```

**\$#** is the string "1", and the standard input is appended (copied) onto the end of *file* using the **cat(1)** command.

```
append file1 file2
```

appends the contents of *file1* onto *file2*. If the number of arguments supplied to **append** is other than 1 or 2, then a message is printed indicating proper usage.

The general form of the **case** command is

```
case word in
  pattern) command-list;;
  ...
esac
```

The **shell** attempts to match word with each pattern in the order in which the patterns appear. If a match is found, the associated **command-list** is executed and execution of the **case** is complete. Since **\*** is the pattern that matches any string, it can be used for the default case.

**Caution:** *No check is made to ensure that only one pattern matches the case argument.*

The first match found defines the set of commands to be executed. In the example below, the commands following the second **"\*"** will never be executed since the first **"\*"** executes everything it receives.

```
case $# in
  *) ... ;;
  *) ... ;;
esac
```

Another example of the use of the **case** construction is to distinguish between different forms of an argument. The following example is a fragment of a **cc(1)** command.

```
for i
do
  case $i in
```



```

-[ocs]) ... ;;
-*) echo 'unknown flag $i' ;;
*.c) /lib/c0 $i ... ;;
*) echo \ 'unexpected argument $i\' ;;
    esac
done

```

To allow the same commands to be associated with more than one pattern, the **case** command provides for alternative patterns separated by a **|**. For example,

```

case $i in
  -x|-y) ...
esac

```

is equivalent to

```

case $i in
  -[xy]) ...
esac

```

The usual quoting conventions apply so that

```

case $i in
  \?) ...

```

will match the character **?**.

### C. Here Documents

The **shell** procedure *tel* described under "A. Control Flow—for" in this section uses the file */usr/lib/telnos* to supply the data for **grep(1)**. An alternative is to include this data within the **shell** procedure as a *here document*, as in,

```

for i
do
  grep $i <<!
  ...
  fred mh0123
  bert mh0789
  ...
!
done

```

In this example, the **shell** takes the lines between **<<!** and **!** as the standard input for **grep(1)**. The string **"!"** is arbitrary. The document is being terminated by a line that consists of the string following **<<**.

Parameters are substituted in the document before it is made available to **grep(1)** as illustrated by the following procedure called *edg*.

```

ed $3 <<%
g/$1/s//$2/g
w
%

```

The call



```
edg string1 string2 file
```

is then equivalent to the command

```
ed file <<%
g/string1/s//string2/g
w
%
```

and changes all occurrences of "string1" in *file* to "string2". Substitution can be prevented using \ to quote the special character \$ as in

```
ed $3 <<+
1,\$s/$1/$2/g
w
+
```

[This version of *edg* is equivalent to the first except that *ed*(1) will print a ? if there are no occurrences of the string \$1.] Substitution within a *here* document may be prevented entirely by quoting the terminating string, for example,

```
grep $i <<\#
...
#
```

The document is presented without modification to *grep*. If parameter substitution is not required in a *here* document, this latter form is more efficient.

#### D. Shell Variables

The **shell** provides string-valued variables. Variable names begin with a letter and consist of letters, digits, and underscores. Variables may be given values by writing

```
user=fred box=m000 acct=mh0000
```

which assigns values to the variables *user*, *box*, and *acct*. A variable may be set to the null string by entering

```
null=
```

The value of a variable is substituted by preceding its name with \$; for example,

```
echo $user
```

will echo *fred*.

Variables may be used interactively to provide abbreviations for frequently used strings. For example,

```
b=/usr/fred/bin
mv file $b
```

will move the *file* from the current directory to the directory */usr/fred/bin*. A more general notation is available for parameter (or variable) substitution, as in,

```
echo ${user}
```



which is equivalent to

```
echo $user
```

and is used when the parameter name is followed by a letter or digit. For example,

```
tmp=/tmp/ps
ps a >${tmp}a
```

will direct the output of `ps(1)` to the file `/tmp/psa`, whereas,

```
ps a >$tmpa
```

would cause the value of the variable `tmpa` to be substituted.

Except for `$?`, the following are set initially by the **shell**.

- |             |                                                                                                                                                                                                                                                                                                       |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>\$?</b>  | The exit status (return code) of the last command executed as a decimal string. Most commands return a zero exit status if they complete successfully; otherwise, a nonzero exit status is returned. Testing the value of return codes is dealt with later under <b>if</b> and <b>while</b> commands. |
| <b>\$#</b>  | The number of positional parameters in decimal. Used, for example, in the <b>append</b> command to check the number of parameters.                                                                                                                                                                    |
| <b>\$\$</b> | The process number of this <b>shell</b> (in decimal). Since process numbers are unique among all existing processes, this string is frequently used to generate unique temporary file names. For example,<br><pre>ps a &gt;/tmp/ps\$\$ ... rm /tmp/ps\$\$</pre>                                       |
| <b>\$_</b>  | The process number of the last process run in the background (in decimal).                                                                                                                                                                                                                            |
| <b>\$-</b>  | The current <b>shell</b> flags, such as <b>-x</b> and <b>-v</b> .                                                                                                                                                                                                                                     |

Some variables have a special meaning to the **shell** and should be avoided for general use.

- |               |                                                                                                                                                                                                                                                                                                                                                                                                |
|---------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>\$MAIL</b> | When used interactively, the <b>shell</b> looks at the file specified by this variable before it issues a prompt. If the specified file has been modified since it was last looked at, the <b>shell</b> prints the message "you have mail" before prompting for the next command. This variable is typically set in the file <code>.profile</code> in the user's login directory. For example: |
|---------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

```
MAIL=/usr/mail/fred
```

- |               |                                                                                                                                                                                                                               |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>\$HOME</b> | The default argument for the <code>cd(1)</code> command. The current directory is used to resolve file name references that do not begin with a <code>/</code> and is changed using the <code>cd</code> command. For example, |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

```
cd /usr/fred/bin
```

makes the current directory `/usr/fred/bin`. Then

```
cat wn
```



will print on the terminal the file *wn* in this directory. The command `cd(1)` with no argument is equivalent to

```
cd $HOME
```

This variable is also typically set in the user's login profile.

**\$PATH** A list of directories containing commands (the *search path*). Each time a command is executed by the **shell**, a list of directories is searched for an executable file. If **\$PATH** is not set, the current directory, */bin*, and */usr/bin* are searched by default. Otherwise, **\$PATH** consists of directory names separated by *:*. For example,

```
PATH=:/usr/fred/bin:/bin:/usr/bin
```

specifies that the current directory (the null string before the first *:*), */usr/fred/bin*, */bin*, and */usr/bin* are to be searched in that order. In this way, individual users can have their own "private" commands that are accessible independently of the current directory. If the command name contains a */*, this directory search is not used; a single attempt is made to execute the command.

**\$PS1** The primary **shell** prompt string, by default, "\$ ".

**\$PS2** The **shell** prompt when further input is needed, by default, "> ".

**\$IFS** The set of characters used by *blank interpretation* (See "D. Evaluation and Quoting" in part "Keyword Parameters").

#### E. Test Command

The **test** command is intended for use by **shell** programs. For example,

```
test -f file
```

returns zero exit status if *file* exists and nonzero exit status otherwise. In general, **test** evaluates a predicate and returns the result as its exit status. Some of the more frequently used **test** arguments are given below [see `test(1)` for a complete specification].

|                           |                                                      |
|---------------------------|------------------------------------------------------|
| <code>test s</code>       | true if the argument <i>s</i> is not the null string |
| <code>test -f file</code> | true if <i>file</i> exists                           |
| <code>test -r file</code> | true if <i>file</i> is readable                      |
| <code>test -w file</code> | true if <i>file</i> is writable                      |
| <code>test -d file</code> | true if <i>file</i> is a directory                   |

#### F. Control Flow—while

The actions of the **for** loop and the **case** branch are determined by data available to the **shell**. A **while** or **until** loop and an **if then else** branch are also provided whose actions are determined by the exit status returned by commands. A **while** loop has the general form

```
while command-list1
do
    command-list2
done
```



The value tested by the **while** command is the exit status of the last simple command following **while**. Each time round the loop, *command-list1* is executed; if a zero exit status is returned, then *command-list2* is executed; otherwise, the loop terminates. For example,

```
while test $1
do
    ...
    shift
done
```

is equivalent to

```
for i
do
    ...
done
```

The **shift** command is a shell command that renames the positional parameters **\$2**, **\$3**, ... as **\$1**, **\$2**, ... and loses **\$1**.

Another kind of use for the **while/until** loop is to wait until some external event occurs and then run some commands. In an **until** loop, the termination condition is reversed. For example,

```
until test -f file
do
    sleep 300
done
commands
```

will loop until *file* exists. Each time round the loop, it waits for 5 minutes (300 seconds) before trying again. (Presumably, another process will eventually create the file.)

#### G. Control Flow—if

Also available is a general conditional branch of the form,

```
if command-list
then
    command-list
else
    command-list
fi
```

that tests the value returned by the last simple command following **if**.

The **if** command may be used in conjunction with the **test** command to test for the existence of a file as in

```
if test -f file
then
    process file
else
    do something else
fi
```

An example of the use of **if**, **case**, and **for** constructions is given in "I. The Man Command" in part "Shell Procedures".



A multiple test if command of the form

```

if ...
then
    ...
else
    if ...
    then
        ...
    else
        if ...
        ...
        fi
    fi
fi

```

may be written using an extension of the if notation as,

```

if ...
then
    ...
elif ...
then
    ...
elif ...
...
fi

```

The **touch** command changes the "last modified" time for a list of files. The command may be used in conjunction with **make(1)** to force recompilation of a list of files. The following example is the **touch** command:

```

flag=
for i
do
    case $i in
        -c) flag=N ;;
        *) if test -f $i
            then
                ln $i junk$$
                rm junk$$
            elif test $flag
            then
                echo file \"$i\" does not exist
            else
                >$i
            fi ;;
    esac
done

```

The **-c** flag is used in this command to force subsequent files to be created if they do not already exist. Otherwise, if the file does not exist, an error message is printed. The shell variable *flag* is set to some non-null string if the **-c** argument is encountered. The commands

ln ...; rm ...



make a link to the file and then remove it. The sequence

```
if command1
then
    command2
fi
```

may be written

```
command1 && command2
```

Conversely,

```
command1 || command2
```

executes **command2** only if **command1** fails. In each case, the value returned is that of the last simple command executed.

### Command Grouping

Commands may be grouped in two ways,

```
{ command-list ; }
```

and

```
( command-list )
```

The first form, *command-list* is simply executed. The second form executes *command-list* as a separate process. For example,

```
(cd x; rm junk )
```

executes *rm junk* in the directory *x* without changing the current directory of the invoking **shell**.

The commands

```
cd x; rm junk
```

have the same effect but leave the invoking **shell** in the directory *x*.

### H. Debugging Shell Procedures

The **shell** provides two tracing mechanisms to help when debugging **shell** procedures. The first is invoked within the procedure as

```
set -v
```

(*v* for verbose) and causes lines of the procedure to be printed as they are read. It is useful to help isolate syntax errors. It may be invoked without modifying the procedure by entering

```
sh -v proc ...
```

where *proc* is the name of the **shell** procedure. This flag may be used in conjunction with the **-n** flag which prevents execution of subsequent commands. (Note that typing "**set -n**" at a terminal will render the terminal useless until an end-of-file is typed.)



The command

```
set -x
```

will produce an execution trace with flag `-x`. Following parameter substitution, each command is printed as it is executed. (Try the above at the terminal to see the effect it has.) Both flags may be turned off by typing

```
set -
```

and the current setting of the `shell` flags is available as `$-`.

#### 1. The "man" Command

The following is the `man` command which is used to print sections of the UNIX System User's Manual. It is called by entering

```
man sh
man -t ed
man 2 fork
```

In the first call, the manual section for `sh` is printed. Since no section is specified, section 1 is used. The second call will typeset (`-t` option) the manual section for `ed`. The last call prints the `fork` manual page from section 2 of the manual.

A version of the `man` command follows:

```
cd /usr/man
: 'colon is the comment command'
: 'default is nroff ($N), section 1 ($s)'
N=n s=1
for i
do
  case $i in
    [1-9]*) s=$i ;;
    -t)    N=t ;;
    -n)    N=n ;;
    -*)    echo unknown flag \"$i\" ;;
    *)    if test -f man$s/$i.$s
          then
              ${N}roff man0/${N}aa man$s/$i.$s
          else
              : 'look through all manual sections'
              found=no
              for j in 1 2 3 4 5 6 7 8 9
              do
                  if test -f man$j/$i.$j
                  then man $j $i
                     found=yes
                  fi
              done
              case $found in
                  no) echo '$i: manual page not found'
              esac
          fi ;;
  esac
done
```



## KEYWORD PARAMETERS

Shell variables may be given values by assignment or when a **shell** procedure is invoked. An argument to a **shell** procedure of the form *name=value* that precedes the command name causes *value* to be assigned to *name* before execution of the procedure begins. The value of *name* in the invoking **shell** is not affected. For example,

```
user=fred command
```

will execute **command** with *user* set to *fred*. The **-k** flag causes arguments of the form *name=value* to be interpreted in this way anywhere in the argument list. Such *names* are sometimes called keyword parameters. If any arguments remain, they are available as positional parameters **\$1**, **\$2**, ... .

The **set** command may also be used to set positional parameters from within a procedure. For example,

```
set - *
```

will set **\$1** to the first file name in the current directory, **\$2** to the next, etc. Note that the first argument, **-**, ensures correct treatment when the first file name begins with a **-**.

## A. Parameter Transmission

When a **shell** procedure is invoked, both positional and keyword parameters may be supplied with the call. Keyword parameters are also made available implicitly to a **shell** procedure by specifying in advance that such parameters are to be exported. For example,

```
export user box
```

marks the variables *user* and *box* for export. When a **shell** procedure is invoked, copies are made of all exportable variables for use within the invoked procedure. Modification of such variables within the procedure does not affect the values in the invoking **shell**. It is generally true of a **shell** procedure that it may not modify the state of its caller without explicit request on the part of the caller. (Shared file descriptors are an exception to this rule.)

Names whose value is intended to remain constant may be declared readonly. The form of this command is the same as that of the **export** command,

```
readonly name ...
```

Subsequent attempts to set readonly variables are illegal.

## B. Parameter Substitution

If a **shell** parameter is not set, then the null string is substituted for it. For example, if the variable *d* is not set,

```
echo $d
```

or

```
echo ${d }
```

will echo nothing. A default string may be given as in

```
echo ${d- }
```



which will echo the value of the variable *d* if it is set and "." otherwise. The default string is evaluated using the usual quoting conventions so that

```
echo ${d- '*'}
```

will echo \* if the variable *d* is not set. Similarly,

```
echo ${d-$1}
```

will echo the value of *d* if it is set and the value (if any) of *\$1* otherwise. A variable may be assigned a default value using the notation

```
echo ${d=.}
```

which substitutes the same string as

```
echo ${d-.
```

and if *d* were not previously set, it will be set to the string ".". (The notation `${...=...}` is not available for positional parameters.)

If there is no sensible default, the notation

```
echo ${d?message}
```

will echo the value of the variable *d* if it has one; otherwise, *message* is printed by the **shell** and execution of the **shell** procedure is abandoned. If *message* is absent, a standard message is printed. A **shell** procedure that requires some parameters to be set might start as follows:

```
: ${user?} ${acct?} ${bin?}
```

```
...
```

Colon (:) is a command built in to the **shell** and does nothing once its arguments have been evaluated. If any of the variables *user*, *acct*, or *bin* are not set, the **shell** will abandon execution of the procedure.

### C. Command Substitution

The standard output from a command can be substituted in a similar way to parameters. The command `pwd(1)` prints on its standard output the name of the current directory. For example, if the current directory is */usr/fred/bin.*, the command

```
d='pwd'
```

is equivalent to

```
d=/usr/fred/bin
```

The entire string between grave accents ('...') is taken as the command to be executed and is replaced with the output from the command. The command is written using the usual quoting conventions except that a ' must be escaped using a \. For example,

```
ls `echo "$1"`
```



is equivalent to

```
ls $1
```

Command substitution occurs in all contexts where parameter substitution occurs (including *here* documents), and the treatment of the resulting text is the same in both cases. This mechanism allows string processing commands to be used within *shell* procedures. An example of such a command is **basename** which removes a specified suffix from a string. For example,

```
basename main.c .c
```

will print the string "main". Its use is illustrated by the following fragment from a *cc(1)* command.

```
case $A in
...
*.c)      B='basename $A .c'
...
esac
```

that sets **B** to the part of **\$A** with the suffix **.c** stripped.

Here are some composite examples.

- for *i* in 'ls -t'; do ...

The variable *i* is set to the names of files in time order, most recent first.

- set 'date'; echo \$6 \$2 \$3, \$4

will print, e.g., 1977 Nov 1, 23:59:59

#### D. Evaluation and Quoting

The *shell* is a macro processor that provides parameter substitution, command substitution, and file name generation for the arguments to commands. This section discusses the order in which these evaluations occur and the effects of the various quoting mechanisms.

Commands are parsed initially according to the grammar given in Table 5.A. Before a command is executed, the following substitutions occur:

1. parameter substitution, e.g., **\$user**
2. command substitution, e.g., **'pwd'**

Only one evaluation occurs so that if, for example, the value of the variable *X* is the string "\$y" then

```
echo $X
```

will echo "\$y".

3. blank interpretation

Following the above substitutions, the resulting characters are broken into nonblank words (*blank interpretation*). For this purpose, 'blanks' are the characters of the string "\$IFS". By default, this string



consists of blank, tab, and newline. The null string is not regarded as a word unless it is quoted. For example,

```
echo ''
```

will pass on the null string as the first argument to **echo**, whereas

```
echo $null
```

will call **echo** with no arguments if the variable *null* is not set or set to the null string.

4. file name generation Each word is then scanned for the file pattern characters \*, ?, and [...]; and an alphabetical list of file names is generated to replace the word. Each such file name is a separate argument.

The evaluations just described also occur in the list of words associated with a **for** loop. Only substitution occurs in the *word* used for a **case** branch.

As well as the quoting mechanisms described earlier using \ and '...', a third quoting mechanism is provided using double quotes. Within double quotes, parameter and command substitution occurs; but file name generation and the interpretation of blanks does not. The following characters have a special meaning within double quotes and may be quoted using \.

|    |                                        |
|----|----------------------------------------|
| \$ | parameter substitution                 |
| '  | command substitution                   |
| "  | ends the quoted string                 |
| \  | quotes the special characters \$ ' " \ |

For example,

```
echo "$x"
```

will pass the value of the variable *x* as a single argument to **echo**. Similarly,

```
echo "$@"
```

will pass the positional parameters as a single argument and is equivalent to

```
echo "$1 $2 ..."
```

The notation **\$@** is the same as **\$\*** except when it is quoted. Inputting

```
echo "$@"
```

will pass the positional parameters, unevaluated, to **echo** and is equivalent to

```
echo "$1" "$2" ...
```

The following illustration gives, for each quoting mechanism, the shell metacharacters that are evaluated.



|   | metacharacter |    |   |   |   |   |
|---|---------------|----|---|---|---|---|
|   | \             | \$ | * | ` | " | ' |
| ' | n             | n  | n | n | n | t |
| ` | y             | n  | n | t | n | n |
| " | y             | y  | n | y | t | n |

t = terminator  
 y = interpreted  
 n = not interpreted

In cases where more than one evaluation of a string is required, the built-in command **eval** may be used. For example, if the variable *X* has the value "\$y" and if *y* has the value "pqr", then

```
eval echo $X
```

will echo the string "pqr".

In general, the **eval** command evaluates its arguments (as do all commands) and treats the result as input to the **shell**. The input is read and the resulting command(s) executed. For example,

```
wg='eval who|grep'
$wg fred
```

is equivalent to

```
who|grep fred
```

In this example, **eval** is required since there is no interpretation of metacharacters, such as **|**, following substitution.

#### E. Error Handling

The treatment of errors detected by the **shell** depends on the type of error and on whether the **shell** is being used interactively. An interactive **shell** is one whose input and output are connected to a terminal [as determined by **gtty(2)**]. A **shell** invoked with the **-i** flag is also interactive.

Execution of a command (see also "G. Command Execution") may fail for any of the following reasons:

- Input/output redirection may fail. For example, if a file does not exist or cannot be created.
- The command itself does not exist or cannot be executed.
- The command terminates abnormally, for example, with a "bus error" or "memory fault" signal.
- The command terminates normally but returns a nonzero exit status.

In all of these cases, the **shell** will go on to execute the next command. Except for the last case, an error message will be printed by the **shell**. All remaining errors cause the **shell** to exit from a command procedure. An interactive **shell** will return to read another command from the terminal. Such errors include the following:

- Syntax errors, e.g., if ...then... done



- A signal such as interrupt. The **shell** waits for the current command, if any, to finish execution and then either exits or returns to the terminal
- Failure of any of the built-in commands such as **cd(1)**.

The **shell** flag **-e** causes the **shell** to terminate if any error is detected. The following is a list of the UNIX operating system signals:

|     |                                             |
|-----|---------------------------------------------|
| 1   | hangup                                      |
| 2   | interrupt                                   |
| 3*  | quit                                        |
| 4*  | illegal instruction                         |
| 5*  | trace trap                                  |
| 6*  | IOT instruction                             |
| 7*  | EMT instruction                             |
| 8*  | floating point exception                    |
| 9   | Kill (cannot be caught or ignored)          |
| 10* | bus error                                   |
| 11* | segmentation violation                      |
| 12* | bad argument to system call                 |
| 13  | write on a pipe with no one to read it      |
| 14  | alarm clock                                 |
| 15  | software termination [from <b>kill(1)</b> ] |

The UNIX operating system signals marked with an asterisk "\*" as shown in the list produce a core dump if not caught. However, the **shell** itself ignores quit which is the only external signal that can cause a dump. The signals in this list of potential interest to **shell** programs are 1, 2, 3, 14, and 15.

#### F. Fault Handling

**Shell** procedures normally terminate when an interrupt is received from the terminal. The **trap** command is used if some cleaning up is required, such as removing temporary files. For example,

```
trap 'rm /tmp/ps$$; exit' 2
```

sets a trap for signal 2 (terminal interrupt); and if this signal is received, it will execute the following commands:

```
rm /tmp/ps$$; exit
```



The **exit** is another built-in command that terminates execution of a **shell** procedure. The **exit** is required; otherwise, after the trap has been taken, the **shell** will resume executing the procedure at the place where it was interrupted.

UNIX operating system signals can be handled in one of three ways.

1. They can be ignored, in which case the signal is never sent to the process.
2. They can be caught, in which case the process must decide what action to take when the signal is received.
3. They can be left to cause termination of the process without it having to take any further action.

If a signal is being ignored on entry to the **shell** procedure, for example, by invoking it in the background (see "G. Command Execution"), **trap** commands (and the signal) are ignored.

The use of **trap** is illustrated by this modified version of the **touch** command illustrated below:

```
flag=
trap 'rm -f junk$$; exit' 1 2 3 15
for i
do
    case $i in
        -c)  flag=N ;;
        *)   if test -f $i
              then
                  ln $i junk$$; rm junk$$
              elif test $flag
              then
                  echo file \"$i\" does not exist
              else
                  >$i
              fi ;;
    esac
done
```

The cleanup action is to remove the file *junk\$\$*. The **trap** command appears before the creation of the temporary file; otherwise, it would be possible for the process to die without removing the file.

Since there is no signal 0 in the UNIX operating system, it is used by the **shell** to indicate the commands to be executed on exit from the **shell** procedure.

A procedure may, itself, elect to ignore signals by specifying the null string as the argument to **trap**. The following:

```
trap '' 1 2 3 15
```

is a fragment taken from the **nohup(1)** command which causes the UNIX operating system HANGUP, INTERRUPT, QUIT, and SOFTWARE TERMINATION signals to be ignored both by the procedure and by invoked commands.

Traps may be reset by entering

```
trap 2 3
```

which resets the traps for signals 2 and 3 to their default values. A list of the current values of traps may be obtained by writing

```
trap
```



The **scan** procedure is an example of the use of **trap** where there is no exit in the trap command. The **scan** takes each directory in the current directory, prompts with its name, and then executes commands typed at the terminal until an end of file or an interrupt is received. Interrupts are ignored while executing the requested commands but cause termination when **scan** is waiting for input. The **scan** procedure follows:

```
d='pwd'
for i in *
do
  if test -d $d/$i
  then cd $d/$i
    while echo "$i:"
      trap exit 2
      read x
    do trap : 2; eval $x; done
  fi
done
```

The **read x** is a built-in command that reads one line from the standard input and places the result in the variable **x**. It returns a nonzero exit status if either an end-of-file is read or an interrupt is received.

#### G. Command Execution

To run a command (other than a built-in), the **shell** first creates a new process using the system call **fork(2)**. The execution environment for the command includes input, output, and the states of signals and is established in the child process before the command is executed. The built-in command **exec** is used in rare cases when no fork is required and simply replaces the **shell** with a new command. For example, a simple version of the **nohup** command looks like

```
trap '' 1 2 3 15
exec $*
```

The **trap** turns off the signals specified so that they are ignored by subsequently created commands, and **exec** replaces the **shell** by the command specified.

Most forms of input/output redirection have already been described. In the following, *word* is only subject to parameter and command substitution. No file name generation or blank interpretation takes place so that, for example,

```
echo ... >*.c
```

will write its output into a file whose name is *\*.c*. Input/output specifications are evaluated left to right as they appear in the command. Some input/output specifications are as follows:

- > *word*                    The standard output (file descriptor 1) is sent to the file *word* which is created if it does not already exist.
- >> *word*                   The standard output is sent to file *word*. If the file exists, then output is appended (by seeking to the end); otherwise, the file is created.
- < *word*                    The standard input (file descriptor 0) is taken from the file *word*.
- << *word*                   The standard input is taken from the lines of **shell** input that follow up to but not including a line consisting only of *word*. If *word* is quoted, no interpretation of the document occurs. If *word* is not quoted, parameter and command substitution occur and \ is used to



quote the characters `\`, `$`, `'`, and the first character of *word*. In the latter case, `\newline` is ignored (e.g., quoted strings).

- `>& digit`      The file descriptor *digit* is duplicated using the system call `dup(2)`, and the result is used as the standard output.
- `<& digit`      The standard input is duplicated from file descriptor *digit*.
- `<&-`            The standard input is closed.
- `>&-`            The standard output is closed.

Any of the above may be preceded by a digit in which case the file descriptor created is that specified by the digit instead of the default 0 or 1. For example,

```
... 2>file
```

runs a command with message output (file descriptor 2) directed to *file*. Another example,

```
... 2>&1
```

runs a command with its standard output and message output merged. (Strictly speaking, file descriptor 2 is created by duplicating file descriptor 1; but the effect is usually to merge the two streams.)

The environment for a command run in the background such as

```
list *.c | lpr &
```

is modified in two ways. First, the default standard input for such a command is the empty file `/dev/null`. This prevents two processes (the **shell** and the command), which are running in parallel, from trying to read the same input. Chaos would ensue if this were not the case. For example,

```
ed file &
```

would allow both the editor and the **shell** to read from the same input at the same time.

The other modification to the environment of a background command is to turn off the QUIT and INTERRUPT signals so that they are ignored by the command. This allows these signals to be used at the terminal without causing background commands to terminate. For this reason, the UNIX operating system convention for a signal is that if it is set to 1 (ignored) then it is never changed even for a short time. Note that the **shell** command `trap` has no effect for an ignored signal.

#### H. Invoking the Shell

The following flags are interpreted by the **shell** when it is invoked. If the first character of argument zero is a minus, commands are read from the file `.profile`.

- `-c string`      If the `-c` flag is present, then commands are read from *string*.
- `-s`              If the `-s` flag is present or if no arguments remain, commands are read from the standard input. **Shell** output is written to file descriptor 2.
- `-i`              If the `-i` flag is present or if the **shell** input and output are attached to a terminal [as told by `getty(8)`], this **shell** is interactive. In this case, TERMINATE is ignored (so that `kill 0` does not kill an interactive **shell**, and INTERRUPT is caught and ignored (so that `wait` is interruptible). In all cases, QUIT is ignored by the **shell**.



TABLE 5.A  
GRAMMAR

|                        |                                                                                                                                                                                                                                                                                                                                                                                |
|------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>item:</i>           | <i>word</i><br><i>input-output</i><br><i>name = value</i>                                                                                                                                                                                                                                                                                                                      |
| <i>simple-command:</i> | <i>item</i><br><i>simple-command item</i>                                                                                                                                                                                                                                                                                                                                      |
| <i>command:</i>        | <i>simple-command</i><br><i>( command-list )</i><br><i>{ command-list }</i><br><i>for name do command-list done</i><br><i>for name in word ... do command-list done</i><br><i>while command-list do command-list done</i><br><i>until command-list do command-list done</i><br><i>case word in case-part ... esac</i><br><i>if command-list then command-list else-part fi</i> |
| <i>pipeline:</i>       | <i>command</i><br><i>pipeline   command</i>                                                                                                                                                                                                                                                                                                                                    |
| <i>andor:</i>          | <i>pipeline</i><br><i>andor &amp;&amp; pipeline</i><br><i>andor !! pipeline</i>                                                                                                                                                                                                                                                                                                |
| <i>command-list:</i>   | <i>andor</i><br><i>command-list ;</i><br><i>command-list &amp;</i><br><i>command-list ; andor</i><br><i>command-list &amp; andor</i>                                                                                                                                                                                                                                           |
| <i>input-output:</i>   | <i>&gt; file</i><br><i>&lt; file</i><br><i>&gt; file</i><br><i>&gt;&gt; word</i><br><i>&lt;&lt; word</i>                                                                                                                                                                                                                                                                       |
| <i>file:</i>           | <i>word</i><br><i>&amp; digit</i><br><i>&amp; -</i>                                                                                                                                                                                                                                                                                                                            |
| <i>case-part:</i>      | <i>pattern ) command-list ;</i>                                                                                                                                                                                                                                                                                                                                                |
| <i>pattern:</i>        | <i>word</i><br><i>pattern   word</i>                                                                                                                                                                                                                                                                                                                                           |
| <i>else-part:</i>      | <i>elif command-list then command-list else-part</i><br><i>else command-list</i><br><i>empty</i>                                                                                                                                                                                                                                                                               |
| <i>empty:</i>          |                                                                                                                                                                                                                                                                                                                                                                                |
| <i>word:</i>           | a sequence of nonblank characters                                                                                                                                                                                                                                                                                                                                              |
| <i>name:</i>           | a sequence of letters, digits, or underscores starting with a letter                                                                                                                                                                                                                                                                                                           |
| <i>digit:</i>          | 0 1 2 3 4 5 6 7 8 9                                                                                                                                                                                                                                                                                                                                                            |



TABLE 5.B

## METACHARACTERS AND RESERVED WORDS

(a) *syntactic:*

|     |                            |
|-----|----------------------------|
|     | pipe symbol                |
| &&  | 'andf' symbol              |
|     | 'orf' symbol               |
| ;   | command separator          |
| ::  | case delimiter             |
| &   | background commands        |
| ( ) | command grouping           |
| <   | input redirection          |
| <<  | input from a here document |
| >   | output creation            |
| >>  | output append              |

(b) *patterns:*

|       |                                       |
|-------|---------------------------------------|
| *     | match any character(s) including none |
| ?     | match any single character            |
| [...] | match any of the enclosed characters  |

(c) *substitution:*

|         |                                  |
|---------|----------------------------------|
| \${...} | substitute <b>shell</b> variable |
| '...'   | substitute command output        |

(d) *quoting:*

|       |                                                              |
|-------|--------------------------------------------------------------|
| \     | quote the next character                                     |
| '...' | quote the enclosed characters except for '                   |
| "..." | quote the enclosed characters except for the \$, ', \, and " |

(e) *reserved words:*

if then else elif fi  
case in esac  
for while until do done  
{ } [ ] test



## 6. REMOTE JOB ENTRY (RJE) USER'S GUIDE

### INTRODUCTION

A set of background processes support remote job entry (RJE) from a UNIX operating system to IBM System/360 and /370 host machines (computers). RJE is the communal name for this subsystem.

The UNIX operating system communicates with the IBM Job Entry Subsystem by mimicking an IBM 360 remote multileaving work station. The UNIX System Administrator's Manual under `rje(8)` summarizes its design and operating procedures. The UNIX System User's Manual also contains a description of the `send(1C)` command, which is the primary user method of submitting jobs to the RJE facilities. (In this document, RJE refers to the facilities provided by the UNIX operating system and not to the Remote Job Entry facilities of the IBM HASP or JES2 subsystems). The `rjestat(1C)` command allows the user to monitor the status of RJE and to send operator commands to the host machine.

Throughout this section, each reference of the form `name(1M)`, `name(7)`, or `name(8)` refers to entries in the UNIX System Administrator's Manual. All other references to entries of the form `name(N)`, where "N" is a number (1 through 6) possibly followed by a letter, refer to entry `name` in section N of the UNIX System User's Manual.

This guide is a tutorial overview of RJE and is addressed to the user who needs to know how to use the RJE facilities but does not need to know details of its implementation. The two following parts constitute a general introduction to the RJE facilities. A sample JES2 output listing can be found in Table 6.A.

### GENERAL

The user should have access to a copy of the UNIX System User's Manual. This manual contains a description of the system and includes a section How to Get Started, which introduces the user to UNIX operating system; the user should be familiar with the contents of that section before proceeding with this document.

In order to begin using the RJE facilities, the user need only become familiar with a subset of basic commands. The user must understand the directory structure of the file system and should know something about the attributes of files: see `cd(1)`, `chmod(1)`, `chown(1)`, `cp(1)`, `ln(1)`, `ls(1)`, `mkdir(1)`, `mv(1)`, `rm(1)`. The user must know how to enter, examine, edit, and print out text files: see `cat(1)`, `ed(1)`, `pr(1)`. The user should also know how to communicate with other users and with the system: see `mail(1)`, `mesg(1)`, `who(1)`, `write(1)`. And, finally, the user might have to know how to describe the user terminal to the system: see `ascii(5)`, `stty(1)`, and `tabs(1)`.

### BASIC RJE

#### A. Submitting Jobs

Assume that the user has used the standard text editor, `ed(1)`, to create a file, *jobfile*, that contains the user's job control statements (JCL) and input data. This file should look exactly like a card deck except that alphabetic characters, for convenience, may be in either uppercase or lowercase. Below is an example:

```
$ cat jobfile
//gener job (9999,r740),pgmname,class=x usr=(mylogin,myplace)
//step exec pgm=iebgener
//sysprint dd sysout=a
//sysin dd dummy
//sysut2 dd sysout=a
//sysut1 dd *
      first card of input data
```



```
      .  
      .  
      .  
      last card of input data  
/* (comment card)
```

To submit the job *jobfile* for execution, the user must invoke the **send(1)** command:

```
$ send jobfile
```

The system will reply:

```
10 cards  
Queued as /usr/rje/rd3125
```

Note that **send** tells how many cards it submitted and reports the position that the job *jobfile* has been assigned in the queue of all jobs waiting to be transmitted to the host machine system. Until the transmission of the job actually begins, the user can prevent the job from being transmitted by doing a **chmod 0** on the queued file to make it unreadable. For the above example, transmission could be stopped by entering:

```
chmod 0 /usr/rje/rd3125 .
```

#### B. Job Messages

Two job messages, a job acknowledgment message (not returned by all host machines) and a job ready message, will be returned with the job from the host machine. Two bells, with an interval of one second between the bells, precede each message. The bells should be interpreted by the user as a warning to stop typing on the terminal, so that the imminent message will not be interspersed with the typing input of the user.

When the job *jobfile* is accepted by the host machine, a job number will be assigned to it; and a job acknowledgment message will be generated for this event. This indicates that the job *jobfile* has been scheduled on the host machine. Later, after the job has executed, the job output file can be directed to the UNIX operating system, and a job ready message is returned for this event. The job output file can also be redirected to another device (card punch, etc.) on the host machine. The user will be notified automatically for both of the events. If the user is logged in when RJE detects these events and is permitting messages to be sent to the terminal [see **mesg(1)**], the following two messages will be sent (still using the example above):

##### Job Acknowledgment Message

Two bells \$12:18:42 gener job 384 — — rd3125 acknowledged

##### Job Ready Message

Two bells

12:21:54 gener job 384 — — /a1/user/rje/prnt0 ready

If the user is not logged in when one of these events occurs or does not allow messages to be sent to the terminal, then the notification will be posted to the user via the **mail(1)** command. The user can prevent messages directly by executing the **mesg(1)** command or indirectly by executing another command, such as **pr(1)**, which prohibits messages for as long as it is active. The user may inspect (by invoking the **mail** command) the mail file (*/usr/mail/logname*) at any time for messages that have been diverted. Setting the user **MAIL** variable to the name of the user mail file will cause the **shell** to notify the user when mail arrives. For this example, the mail notice might be as follows:

```
$ mail  
From rje Mon Aug 1 12:20:36 1981
```



```
$12:18:42 gener job 384 -- rd3125 acknowledged
? d
From rje Mon Aug 1 12:21:55 1981
12:21:54 gener job 384 -- /a1/user/rje/prnt0 ready
? d
```

The job acknowledgment message performs two functions. First, the message confirms the fact that the job has been scheduled for execution. Second, the message assigns a number to the job in such a way that the number and the name together will uniquely identify the job for a period of time on the host machine.

The output ready message provides the name of a UNIX file (prnt0) into which output has been written and identifies the job to which the output belongs [see ls(1)]:

```
$ ls -l prnt0
-r--r--r-- 1 rje      1184 Aug 1 12:21 prnt0
```

Note that "rje" retains ownership of the output file and allows only the user to have read access to it (no group id name is listed). The user should inspect the file, perhaps extract some information from it, and then promptly delete the file thus [see rm(1)]:

```
$ rm -f prnt0
```

#### C. Output File Retention

The retention of machine-generated files, such as RJE output files, is discouraged. It is the responsibility of the user to remove files from the RJE directory to conserve memory space. The RJE output files may be truncated if the output file exceeds a set limit. This limit is tunable by the system administrator. The part of the output file that goes beyond the currently set limit will be discarded, with no provision for retrieval. If the output file is truncated, a warning message will appear in the truncated output file as follows:

```
Two bells
12:21:54 gener job 384 -- /a1/user/rje/prnt0 ready (truncated)
```

The user should also be aware that RJE attempts to keep a set number of blocks free on any file system it uses. This number is also tunable by the system administrator. Warning messages or suspension of certain functions will occur as this limit is approached.

#### D. Examining Output Files

The most elementary way to examine the output file is to cat the file to the user terminal. The appendix illustrates the result of listing the output file of our sample job in this way. Because the UNIX operating system does not have high volume printing capability, any large listings should be routed to the host machine.

The structure of an output file (listing) will generally conform to the following sequence:

```
HASP log
jcl information
data sets
HASP end
```

Normally "burst" pages will not be present. Single, double, and triple spacing is reflected in the output file, but other "forms" controls, such as the skip to the top of a new page, are suppressed. Page boundaries are indicated by the presence of a blank (space character) at the end of the last line of each page.

The big file scanner bfs(1) or the standard text editor ed(1) provides a more flexible method than cat(1) for examining printed output; bfs can handle files of any size and is more efficient than ed for scanning files.



The RJE facility is also capable of receiving punched card output data as formatted files [see **pnch(4)**]; this format allows an exact representation of an arbitrary card deck to be stored in the UNIX operating system. However, there are few commands that can be used to manipulate these files. The user should route or redirect the punched output file to one of the host machine output devices.

## SEND COMMAND

The **send(1C)** command is capable of invoking more general processing than has been indicated previously in Basic RJE. The **send** command will concatenate a sequence of files to create a single job stream. This allows files of JCL and files of data to be maintained separately in the UNIX operating system. In addition, it recognizes any line of an input file that begins with the character '~' as being a **control** line that can call for the inclusion, inside the current file, of some other file. This allows the user to send a top level skeleton file that "pulls" or calls in subordinate files as needed. Some of these may be "virtual" files that actually consist of the output of UNIX commands or **shell** procedures. Furthermore, the **send** command is able to collect input data directly from a terminal and can be instructed to prompt the terminal user for the required data.

Each source of input can contain a format specification that determines such things as how to expand tabs and how long can an input line be. File specification **fspec(4)** explains how to define such formats. When properly instructed, **send** will also replace arbitrarily defined keywords by other text strings or by Extended Binary Coded Decimal Interchange Code (EBCDIC) character codes. (These two substitution facilities are useful in other applications besides RJE; for that reason, the **send** command may be invoked under the name **gath(1C)** to produce the standard output without submitting an RJE job).

Two options of the **send** command that all users should be acquainted with are—first, the ability to specify to which host machine a job is to be submitted and, second, a flag that guarantees that a job will be transmitted to the host machine in the order of submission (relative to other jobs submitted with the same flag). To execute the above sample job on a host machine cited to RJE as A, issue the command:

```
$ send A jobfile
```

When a host machine is not explicitly cited, **send** makes a reasonable choice.

To ensure that a job will be transmitted in the order of submission, set the **-x** flags of **send** by entering:

```
$ send A -x jobfile
```

This flag should be used sparingly. The complete list of arguments and flags that control the execution of **send** are defined under the **send(1C)** command.

## JOB STREAM

It is assumed that the job stream submitted as the result of a single execution of **send** consists of a single "job", i.e., the file that is queued for transmission should contain one JOB card near the beginning and no other job cards. A priority control card may legitimately precede the JOB card. The JOB card must conform to the standards of the host machine. A typical JOB card has the following format:

### JOB Card Format

```
//name job (acct[,...]),pgmrname[,keywds=?] [usr=...]
```

## USER FIELD SPECIFICATION

### A. Options

The user field "usr=..." (specified on the job card, comment card, or input data card) is required for any print or punch output file that is to be returned to the UNIX operating system user. The format of this field is:

```
usr=(login,place,[level])
```



where *login* is the UNIX login name of the user, *level* is the desired level of notification, and the *place* value is defined in cases A through E below:

- A. If *place* is the name of a directory (writable by others), then the output file is placed there as a unique *prnt* or *pnch* file. The permission mode of the file will be 454.
- B. If *place* is the name of an existing, writable (by others), nonexecutable (by others) file, then the output file replaces it. The permission mode of the file will be 454.
- C. If *place* is the name of a nonexistent file in a writable (by others) directory, then the output file is placed there. The permission mode of the file will be 454.
- D. If *place* is the name of an executable (by others) file, then the RJE output is set up as standard input to *place* and *place* is executed. Five string arguments are passed to *place*. For example, if *place* is a *shell* procedure, the following arguments are passed as \$1 ... \$5:

- 1. Flag indicating whether file space is scarce in the file system where *place* resides. A 0 indicates that space is **not** scarce; 1 indicates space is scarce.
- 2. Job name.
- 3. Name of programmer.
- 4. Job number.
- 5. Login name from the "usr=..." field specifications.

A ":" is passed if a value is not present. The current directory for the execution of *place* will be set to the directory containing *place*. The environment [see *environ*(5)] will contain values for **LOGNAME** and **HOME** based on the login name from the "usr=..." specification and a value for **TZ** (time zone). Since the login name supplied in the "usr=..." field specification cannot be believed for security purposes, the UID will be set to a reserved value.

- E. In all other cases, the output will be discarded.

The *place* value must not be a full pathname, unless it refers to an executable file (refer to case D above). For cases A, B, and C above (and case D, if a full pathname is not supplied), the name of the login directory of the user will be used to form a full pathname.

The "usr=..." field specification may occur anywhere within the first 100 card images sent and within the first 200 output images received by the UNIX operating system. The only restriction is that it be contained completely on a single line or card image. Therefore, the "usr=..." field may be placed on a JOB card or comment card. The field may also be passed as data on an input data card.

If the output files are to be redirected from the host machine to another UNIX system machine, a "usr=..." specification field card, if not already present, must be supplied by the user. This can be done by placing a job step that creates this card before the output steps. The remote UNIX system machine will handle the output files as specified on the "usr=..." specification field card.



## B. RJE Message Level of Notification

Messages generated by RJE or returned from the host machine are assigned a level of importance ranging from 1 to 9. The levels in use are:

- 3 transmittal assurance
- 5 job acknowledgment
- 6 output ready message.

The optional **level** value of the "**usr=...**" field specification must be a 1- or 2-digit code of the form **mw**. A message from the host machine with importance **x** (where **x** comes from the above list) is compared with each digit of the two decimal digits in **level**. If **x** is greater than or equal to **w** and if the user is logged in and is accepting messages, the message will be written to the terminal of the user. Otherwise, if **x** is greater than or equal to **m**, the message will be mailed to the user. In all other cases, the message will be discarded. The default **level** is **54**. The user should specify level 1 if it is desirable to receive complete notification and level 59 to divert the last two messages in the above list to the user mailbox.

## MONITORING RJE

The RJE facility is designed to be an autonomous facility that does not require manual supervision. The RJE facility is initiated automatically by the UNIX operating system reboot procedures and continues in execution until the system is shut down. Experience has shown RJE to be reasonably robust, although it is vulnerable to system crashes and reconfigurations.

Users may assume that when the UNIX operating system is operating, the RJE facilities will also be available. This implies more than an ability to execute the **send(1C)** command, which should be available at all times; it means that queued jobs should be submitted to the host machine for execution and their output returned to the UNIX operating system. If a user cannot obtain any throughput from the RJE facility, the user should so advise the UNIX administrator.

The **rjestat(1C)** command, invoked with no arguments, will report the status of all RJE links for which a given UNIX operating system is configured. This command sometimes will also print a message of the day from RJE.

```
$ rjestat
RJE to B operating normally.
RJE to A is down, reason: IBM not responding.
```

A host machine may be reported not to be responding to RJE because the host machine is down. It may be down because the host machine operator failed to initialize the associated line. It may be down also because of a communications hardware failure.

The **rjestat** command also has the ability to send operator commands to the host machine and to retrieve the responses generated by the commands. Refer to **rjestat(1C)** command page for a more complete description of this command.



TABLE 6.A  
SAMPLE JES2 OUTPUT LISTING

```

$ cat rje/prnt0
14:40:31 JOB 384 $HASP373 GENER STARTED - INIT 26 - CLASS X - SYS RRMA
14:40:32 JOB 384 $HASP395 GENER ENDED

—— JES2 JOB STATISTICS ——
1 AUG 81 JOB EXECUTION DATE
      54 CARDS READ
      76 SYSOUT PRINT RECORDS
      0 SYSOUT PUNCH RECORDS
    0.01 MINUTES EXECUTION TIME
1      //GENER JOB (9999,R740),PGMRNAME,CLASS = X
      ***   USR=(MYLOGIN,MYPLACE)
2      //IEBGENER EXEC PGM=IEBGENER
3      //SYSPRINT DD DUMMY
4      //SYSIN DD DUMMY
5      //SYSUT2 DD SYSOUT=A
6      //SYSUT1 DD *
      //
IEF236I  ALLOC. FOR GENER IEBGENER
IEF237I  DMY ALLOCATED TO SYSPRINT
IEF237I  DMY ALLOCATED TO SYSIN
IEF237I  JES ALLOCATED TO SYSUT2
IEF237I  JES ALLOCATED TO SYSUT1
IEF142I  GENER IEBGENER - STEP WAS EXECUTED - COND CODE 0000
IEF285I  JES2.JOB0384.S00102
IEF285I  JES2.JOB0384.SI0101
IEF373I  STEP /IEBGENER/ START 77242.1440
IEF374I  STEP /IEBGENER/ STOP 7742.1440 CPU    OMIN 00.13SEC SRB    OMIN 00.01SEC VIRT 36K SYS 188K

*****  SERVICE UNITS=0000174    SERVICE RATE=0000268 SERVICE UNITS/SECOND
*****  PERFORMANCE GROUP=005
*****  EXCP COUNT BY UNIT ADDRESS
IEF375I  JOB /GENER / START 77242.1440
IEF376I  JOB /GENER / STOP 7742.1440 CPU    OMIN 00.13SEC SRB    OMIN 00.01SEC

*****  SERVICE UNITS=0000174    SERVICE RATE=0000268 SERVICE UNITS/SECOND
*****  APPROXIMATE PROCESSING TIME=    .01 MINUTES
*****  EXCPS=000000000
*****  PROJECTED CHARGES=    .01
      first line of data
.
.
      last line of data
*OS/VS2  REL 3.7 JES2*  END  JOBNAME=GENER  BIN=R740  JOB #=384  PGMRNAME
*OS/VS2  REL 3.7 JES2*  END  JOBNAME=GENER  BIN=R740  JOB #=384  PGMRNAME
*OS/VS2  REL 3.7 JES2*  END  JOBNAME=GENER  BIN=R740  JOB #=384  PGMRNAME
$ rm -f rje/prnt0

```



NOTES



## 7. SOURCE CODE CONTROL SYSTEM USER'S GUIDE

### GENERAL

The Source Code Control System (SCCS) is a collection of the UNIX operating system commands which help individuals or projects control and account for changes to files of text. The source code and documentation of software systems are typical examples of files of text to be changed. The SCCS is a collection of programs that run under the UNIX operating system. It is convenient to conceive of SCCS as a custodian of files. The SCCS provides facilities for the following:

- Storing files of text
- Retrieving particular versions of the files
- Controlling updating privileges to files
- Identifying the version of a retrieved file
- Recording when, where, and why the change was made and who made each change to a file.

These types of facilities are important when programs and documentation undergo frequent changes because of maintenance and/or enhancement work. It is often desirable to regenerate the version of a program or document as it existed before changes were applied to it. This can be done by keeping copies (on paper or other media), but this method quickly becomes unmanageable and wasteful as the number of programs and documents increases. The SCCS provides an attractive solution because the original file is stored on disk. Whenever changes are made to the file, the SCCS stores only the changes. Each set of changes is called a "delta".

This section, together with relevant portions of the UNIX System User's Manual of the UNIX operating system, is a complete user's guide to SCCS. The following topics are covered:

- SCCS for Beginners: How to make an SCCS file, how to update it, and how to retrieve a version thereof.
- How Deltas Are Numbered: How versions of SCCS files are numbered and named.
- SCCS Command Conventions: Conventions and rules generally applicable to all SCCS commands.
- SCCS Commands: Explanation of all SCCS commands, with discussions of the more useful arguments.
- SCCS Files: Protection, format, and auditing of SCCS files including a discussion of the differences between using SCCS as an individual and using it as a member of a group or project. The role of a "project SCCS administrator" is introduced.

Neither the implementation of SCCS nor the installation procedure for SCCS are described in this section.

Throughout this section, each reference of the form **name(1M)**, **name(7)**, or **name(8)** refers to entries in the UNIX System Administrator's Manual. All other references to entries of the form **name(N)**, where "N" is a number (1 through 6) possibly followed by a letter, refer to entry **name** in section N of the UNIX System User's Manual.

### SCCS FOR BEGINNERS

It is assumed that the reader knows how to log onto a UNIX operating system, create files, and use the text editor. A number of terminal-session fragments are presented. All of them should be tried since the best way to learn SCCS is to use it.



To supplement the material in this section, the detailed SCCS command descriptions in the UNIX System User's Manual of the UNIX operating system should be consulted.

#### A. Terminology

Each SCCS file is composed of one or more sets of changes applied to the null (empty) version of the file, with each set of changes usually depending on all previous sets. Each set of changes is called a "delta" and is assigned a name, called the *SCCS IDentification string* (SID). The SID is composed of at most four components. The first two components are the "release" and "level" numbers which are separated by a period. Hence, the first delta (for the original file) is called "1.1", the second "1.2", the third "1.3", etc. The release number can also be changed allowing, for example, deltas "2.1", "3.19", etc. The change in the release number usually indicates a major change to the file.

Each delta of an SCCS file defines a particular version of the file. For example, delta 1.5 defines version 1.5 of the SCCS file, obtained by applying to the null (empty) version of the file the changes that constitute deltas 1.1, 1.2, etc., up to and including delta 1.5 itself, in that order.

#### B. Creating an SCCS File via "admin"

Consider, for example, a file called *lang* that contains a list of programming languages:

```
c
pl/i
fortran
cobol
algol
```

Custody of the *lang* file can be given to SCCS. The following `admin(1)` command (used to "administer" SCCS files) creates an SCCS file and initializes delta 1.1 from the file *lang*.

```
admin -ilang s.lang
```

All SCCS files must have names that begin with "s.", hence, *s.lang*. The `-i` keyletter, together with its value *lang*, indicates that `admin` is to create a new SCCS file and "initialize" the new SCCS file with the contents of the file *lang*. This initial version is a set of changes (delta 1.1) applied to the null SCCS file.

The `admin` command replies

```
No id keywords (cm7)
```

This is a warning message (which may also be issued by other SCCS commands) that is to be ignored for the purposes of this section. Its significance is described under the `get(1)` command in the part SCCS Commands. In the following examples, this warning message is not shown, although it may actually be issued by the various commands. The file *lang* should now be removed (because it can be easily reconstructed using the `get` command) as follows:

```
rm lang
```

#### C. Retrieving a File via "get"

The *lang* file can be reconstructed by using the following `get` command:

```
get s.lang
```

The command causes the creation (retrieval) of the latest version of file *s.lang* and prints the following messages:

```
1.1
5 lines
```



This means that **get** retrieved version 1.1 of the file, which is made up of five lines of text. The retrieved text is placed in a file whose name is formed by deleting the "s." prefix from the name of the SCCS file. Hence, the file *lang* is created.

The "get s.lang" command simply creates the file *lang* (read-only) and keeps no information regarding its creation. On the other hand, in order to be able to subsequently apply changes to an SCCS file with the **delta(1)** command, the **get** command must be informed of your intention to do so. This is done as follows:

```
get -e s.lang
```

The **-e** keyletter causes **get** to create a file *lang* for both reading and writing (so it may be edited) and places certain information about the SCCS file in another new file. The new file, called the p-file, will be read by the **delta** command. The **get** command prints the same messages as before except that the SID of the version to be created through the use of **delta** is also issued. For example:

```
get -e s.lang
1.1
new delta 1.2
5 lines
```

The file *lang* may now be changed, for example, by:

```
ed lang
27
$a
snobol
ratfor
.
w
41
q
```

#### D. Recording Changes via "delta"

In order to record within the SCCS file the changes that have been applied to *lang*, execute the following command:

```
delta s.lang
```

Delta prompts with:

```
comments?
```

the response to which should be a description of why the changes were made. For example:

```
comments? added more languages
```

The **delta** command then reads the p-file and determines what changes were made to the file *lang*. The **delta** command does this by doing its own **get** to retrieve the original version and by applying the **diff(1)** command to the original version and the edited version.

When this process is complete, at which point the changes to *lang* have been stored in *s.lang*, **delta** outputs:

```
1.2
2 inserted
```



```
0 deleted
5 unchanged
```

The number "1.2" is the name of the delta just created, and the next three lines of output refer to the number of lines in the file *s.lang*.

#### E. Additional Information About "get"

As shown in the previous example, the command

```
get s.lang
```

retrieves the latest version (now 1.2) of the file *s.lang*. This is done by starting with the original version of the file and successively applying deltas (the changes) in order until all have been applied.

In the example chosen, the following commands are all equivalent:

```
get s.lang
get -r1 s.lang
get -r1.2 s.lang
```

The numbers following the *-r* keyletter are SIDs. Note that omitting the level number of the SID (as in "get -r1 s.lang") is equivalent to specifying the highest level number that exists within the specified release. Thus, the second command requests the retrieval of the latest version in release 1, namely 1.2. The third command specifically requests the retrieval of a particular version, in this case, also 1.2.

Whenever a truly major change is made to a file, the significance of that change is usually indicated by changing the release number (first component of the SID) of the delta being made. Since normal automatic numbering of deltas proceeds by incrementing the level number (second component of the SID), the user must indicate to SCCS the need to change the release number. This is done with the *get* command:

```
get -e -r2 s.lang
```

Because release 2 does not exist, *get* retrieves the latest version *before* release 2. The *get* command also interprets this as a request to change the release number of the delta the user desires to create to 2, thereby causing it to be named 2.1, rather than 1.3. This information is conveyed to *delta* via the p-file. The *get* command then outputs

```
1.2
new delta 2.1
7 lines
```

which indicates that version 1.2 has been retrieved and that 2.1 is the version *delta* will create. If the file is now edited, for example, by:

```
ed lang
41
/cobol/d
w
35
q
```

and *delta* executed:

```
delta s.lang
comments? deleted cobol from list of languages
```



the user will see by delta's output that version 2.1 is indeed created:

```
2.1
0 inserted
1 deleted
6 unchanged
```

Deltas may now be created in release 2 (deltas 2.2, 2.3, etc.), or another new release may be created in a similar manner. This process may be continued as desired.

#### F. The "help" Command

If the command:

```
get abc
```

is executed, the following message will be output:

```
ERROR [abc]: not an SCCS file (col)
```

The string "col" is a code for the diagnostic message and may be used to obtain a fuller explanation of that message by use of the help(1) command:

```
help col
```

This produces the following output:

```
col:
"not an SCCS file"
A file that you think is an SCCS file does not begin with the characters "s".
```

Thus, **help** is a useful command to use whenever there is any doubt about the meaning of an SCCS message. Detailed explanations of almost all SCCS messages may be found in this manner.

#### DELTA NUMBERING

It is convenient to conceive of the deltas applied to an SCCS file as the nodes of a tree in which the root is the initial version of the file. The root delta (node) is normally named "1.1" and successor deltas (nodes) are named "1.2", "1.3", etc. The components of the names of the deltas are called the "release" and the "level" numbers, respectively. Thus, normal naming of successor deltas proceeds by incrementing the level number, which is performed automatically by SCCS whenever a delta is made. In addition, the user may wish to change the release number when making a delta to indicate that a major change is being made. When this is done, the release number also applies to all successor deltas unless specifically changed again. Thus, the evolution of a particular file may be represented as in Fig. 7.1.



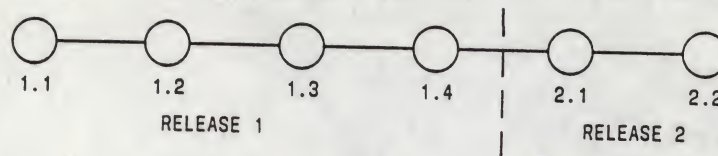


Fig. 7.1 — Evolution of an SCCS File

Such a structure may be termed the “trunk” of the SCCS tree. Figure 7.1 represents the normal sequential development of an SCCS file in which changes that are part of any given delta are dependent upon all the preceding deltas.

However, there are situations in which it is necessary to cause a branching in the tree in that changes applied as part of a given delta are not dependent upon all previous deltas. As an example, consider a program which is in production use at version 1.3 and for which development work on release 2 is already in progress. Thus, release 2 may already have some deltas precisely as shown in Fig. 7.1. Assume that a production user reports a problem in version 1.3 and that the nature of the problem is such that it cannot wait to be repaired in release 2. The changes necessary to repair the trouble will be applied as a delta to version 1.3 (the version in production use). This creates a new version that will then be released to the user but will not affect the changes being applied for release 2 (i.e., deltas 1.4, 2.1, 2.2, etc.).

The new delta is a node on a branch of the tree, and its name consists of four components—the release and level numbers, as with trunk deltas, plus the “branch” and “sequence” numbers. The delta name will appear as follows:

release.level.branch.sequence

The branch number is assigned to each branch that is a descendant of a particular trunk delta with the first such branch being 1, the next one 2, etc. The sequence number is assigned, in order, to each delta on a particular branch. Thus, 1.3.1.2 identifies the second delta of the first branch that derives from delta 1.3. This is shown in Fig. 7.2.

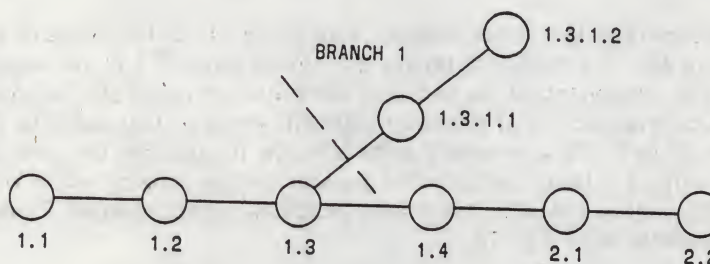


Fig. 7.2 — Tree Structure with Branch Deltas

The concept of branching may be extended to any delta in the tree. The naming of the resulting deltas proceeds in the manner just illustrated.

Two observations are of importance with regard to naming deltas. First, the names of trunk deltas contain



exactly two components, and the names of branch deltas contain exactly four components. Second, the first two components of the name of branch deltas are always those of the ancestral trunk delta, and the branch component is assigned in the order of creation of the branch independently of its location relative to the trunk delta. Thus, a branch delta may always be identified as such from its name. Although the ancestral trunk delta may be identified from the branch delta's name, it is not possible to determine the entire path leading from the trunk delta to the branch delta. For example, if delta 1.3 has one branch emanating from it, all deltas on that branch will be named 1.3.1.n. If a delta on this branch then has another branch emanating from it, all deltas on the new branch will be named 1.3.2.n (see Fig. 7.3). The only information that may be derived from the name of delta 1.3.2.2 is that it is the chronologically second delta on the chronologically second branch whose trunk ancestor is delta 1.3. In particular, it is not possible to determine from the name of delta 1.3.2.2 all the deltas between it and trunk ancestor 1.3.

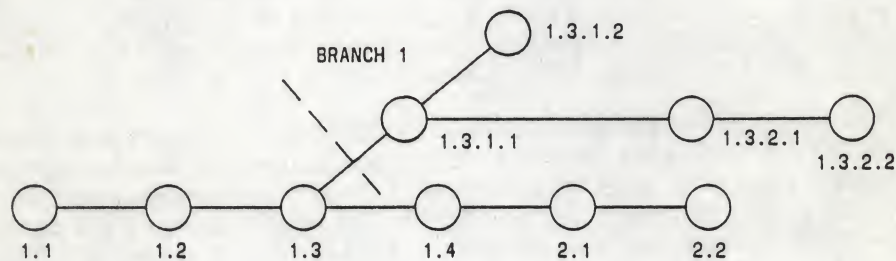


Fig. 7.3 — Extending the Branching Concept

It is obvious that the concept of branch deltas allows the generation of arbitrarily complex tree structures. Although this capability has been provided for certain specialized uses, it is strongly recommended that the SCCS tree be kept as simple as possible because comprehension of its structure becomes extremely difficult as the tree becomes more complex.

### SCCS COMMAND CONVENTIONS

This part discusses the conventions and rules that apply to SCCS commands. These rules and conventions are generally applicable to all SCCS commands with exceptions indicated. The SCCS commands accept two types of arguments:

- keyletter arguments
- file arguments.

Keyletter arguments (hereafter called simply "keyletters") begin with a minus sign (-), followed by a lower-case alphabetic character, and in some cases, followed by a value. These keyletters control the execution of the command to which they are supplied.

File arguments (which may be names of files and/or directories) specify the file(s) that the given SCCS command is to process. Naming a directory is equivalent to naming all the SCCS files within the directory. Non-SCCS files and unreadable files [because of permission modes via `chmod(1)`] in the named directories are silently ignored.

In general, file arguments may not begin with a minus sign. However, if the name "-" (a lone minus sign) is specified as an argument to a command, the command reads the standard input for lines and takes each line



as the name of an SCCS file to be processed. The standard input is read until end-of-file. This feature is often used in pipelines with, for example, the `find(1)` or `ls(1)` commands. Again, names of non-SCCS files and of unreadable files are silently ignored.

All keyletters specified for a given command apply to all file arguments of that command. All keyletters are processed before any file arguments with the result that the placement of keyletters is arbitrary (i.e., keyletters may be interspersed with file arguments). File arguments, however, are processed left to right. Somewhat different argument conventions apply to the `help(1)`, `what(1)`, `sccsdiff(1)`, and `val(1)` commands.

Certain actions of various SCCS commands are controlled by flags appearing in SCCS files. Some of these flags are discussed in this part. For a complete description of all such flags, see the `admin(1)` section in the UNIX System User's Manual of the UNIX operating system.

The distinction between the real user [see `passwd(1)`] and the effective user of a UNIX operating system is of concern in discussing various actions of SCCS commands. For the present, it is assumed that both the real user and the effective user are one and the same (i.e., the user who is logged into a UNIX operating system). This subject is discussed further in the SCCS Files part.

All SCCS commands that modify an SCCS file do so by writing a temporary copy, called the x-file, which ensures that the SCCS file will not be damaged should processing terminate abnormally. The name of the x-file is formed by replacing the "s." of the SCCS file name with "x.". When processing is complete, the old SCCS file is removed and the x-file is renamed to be the SCCS file. The x-file is created in the directory containing the SCCS file, given the same mode [see `chmod(1)`] as the SCCS file, and owned by the effective user.

To prevent simultaneous updates to an SCCS file, commands that modify SCCS files create a lock-file, called the z-file, whose name is formed by replacing the "s." of the SCCS file name with "z.". The z-file contains the process number of the command that creates it, and its existence is an indication to other commands that the SCCS file is being updated. Thus, other commands that modify SCCS files will not process an SCCS file if the corresponding z-file exists. The z-file is created with mode 444 (read-only) in the directory containing the SCCS file and is owned by the effective user. This file exists only for the duration of the execution of the command that creates it. In general, users can ignore x-files and z-files. The files may be useful in the event of system crashes or similar situations.

SCCS commands produce diagnostics (on the diagnostic output) of the form:

ERROR [name-of-file-being-processed]: message text (code)

The code in parentheses may be used as an argument to the `help(1)` command to obtain a further explanation of the diagnostic message. Detection of a fatal error during the processing of a file causes the SCCS command to terminate processing of that file and to proceed with the next file, in order, if more than one file has been named.

## SCCS COMMANDS

This part describes the major features of all the SCCS commands. Detailed descriptions of the commands and of all their arguments are given in the UNIX System User's Manual of the UNIX operating system and should be consulted for further information. The discussion below covers only the more common arguments of the various SCCS commands.

The commands follow in approximate order of importance. The following is a summary of all the SCCS commands and of their major functions:

|                       |                                                                                 |
|-----------------------|---------------------------------------------------------------------------------|
| <code>get(1)</code>   | Retrieves versions of SCCS files.                                               |
| <code>delta(1)</code> | Applies changes (deltas) to the text of SCCS files, i.e., creates new versions. |



|                         |                                                                                                                                                                                                            |
|-------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>admin(1)</code>   | Creates SCCS files and applies changes to parameters of SCCS files.                                                                                                                                        |
| <code>prs(1)</code>     | Prints portions of an SCCS file in user specified format.                                                                                                                                                  |
| <code>help(1)</code>    | Gives explanations of diagnostic messages.                                                                                                                                                                 |
| <code>rmDEL(1)</code>   | Removes a delta from an SCCS file; allows the removal of deltas that were created by mistake.                                                                                                              |
| <code>cdc(1)</code>     | Changes the commentary associated with a delta.                                                                                                                                                            |
| <code>what(1)</code>    | Searches any UNIX operating system file(s) for all occurrences of a special pattern and prints out what follows it; is useful in finding identifying information inserted by the <code>get</code> command. |
| <code>scsdiff(1)</code> | Shows the differences between any two versions of an SCCS file.                                                                                                                                            |
| <code>comb(1)</code>    | Combines two or more consecutive deltas of an SCCS file into a single delta; often reduces the size of the SCCS file.                                                                                      |
| <code>val(1)</code>     | Validates an SCCS file.                                                                                                                                                                                    |

#### A. The "get" Command

The `get(1)` command creates a text file that contains a particular version of an SCCS file. The particular version is retrieved by beginning with the initial version and then applying deltas, in order, until the desired version is obtained. The created file is called the g-file. The g-file name is formed by removing the "s." from the SCCS file name. The g-file is created in the current directory and is owned by the real user. The mode assigned to the g-file depends on how the `get` command is invoked.

The most common invocation of `get` is:

```
get s.abc
```

which normally retrieves the latest version on the trunk of the SCCS file tree and produces (for example) on the standard output:

```
1.3
67 lines
No id keywords (cm7)
```

which indicates that:

1. Version 1.3 of file "s.abc" was retrieved (1.3 is the latest trunk delta).
2. This version has 67 lines of text.
3. No ID keywords were substituted in the file.

The generated g-file (file "abc") is given mode 444 (read-only) since this particular way of invoking `get` is intended to produce g-files only for inspection, compilation, etc., and not for editing (i.e., not for making deltas).

In the case of several file arguments (or directory-name arguments), similar information is given for each file processed, but the SCCS file name precedes it. For example:

```
get s.abc s.def
```



produces:

```
s.abc:
1.3
67 lines
No id keywords (cm7)
```

```
s.def:
1.7
85 lines
No id keywords (cm7)
```

#### ID Keywords

In generating a g-file to be used for compilation, it is useful and informative to record the date and time of creation, the version retrieved, the module's name, etc. within the g-file, so this information will appear in a load module when one is eventually created. The SCCS provides a convenient mechanism for doing this automatically. Identification (ID) keywords appearing anywhere in the generated file are replaced by appropriate values according to the definitions of these ID keywords. The format of an ID keyword is an uppercase letter enclosed by percent signs (%). For example:

%I%

is defined as the ID keyword that is replaced by the SID of the retrieved version of a file. Similarly, %H% is defined as the ID keyword for the current date (in the form "mm/dd/yy"), and %M% is defined as the name of the g-file. Thus, executing `get` on an SCCS file that contains the PL/I declaration:

```
DCL ID CHAR(100) VAR INIT('%M% %I% %H%');
```

gives (for example) the following:

```
DCL ID CHAR(100) VAR INIT('MODNAME 2.3 07/07/77');
```

When no ID keywords are substituted by `get`, the following message is issued:

```
No id keywords (cm7)
```

This message is normally treated as a warning by `get`, although the presence of the `i` flag in the SCCS file causes it to be treated as an error. For a complete list of the approximately 20 ID keywords provided, see `get(1)` in the UNIX System User's Manual of the UNIX operating system.

#### Retrieval of Different Versions

Various keyletters are provided to allow the retrieval of other than the default version of an SCCS file. Normally, the default version is the most recent delta of the highest-numbered release on the trunk of the SCCS file tree. However, if the SCCS file being processed has a `d` (default SID) flag, the SID specified as the value of this flag is used as a default. The default SID is interpreted in exactly the same way as the value supplied with the `-r` keyletter of `get`.

The `-r` keyletter is used to specify a SID to be retrieved, in which case the `d` (default SID) flag (if any) is ignored. For example:

```
get -r1.3 s.abc
```



retrieves version 1.3 of file *s.abc* and produces (for example) on the standard output:

```
1.3
64 lines
```

A branch delta may be retrieved similarly:

```
get -r1.5.2.3 s.abc
```

which produces (for example) on the standard output:

```
1.5.2.3
234 lines
```

When a 2- or 4-component SID is specified as a value for the `-r` keyletter (as above) and the particular version does not exist in the SCCS file, an error message results. Omission of the level number, as in:

```
get -r3 s.abc
```

causes retrieval of the trunk delta with the highest level number within the given release if the given release exists. Thus, the above command might output:

```
3.7
213 lines
```

If the given release does not exist, `get` retrieves the trunk delta with the highest level number within the highest-numbered existing release that is lower than the given release. For example, assuming release 9 does not exist in file *s.abc* and that release 7 is actually the highest-numbered release below 9, execution of:

```
get -r9 s.abc
```

might produce:

```
7.6
420 lines
```

which indicates that trunk delta 7.6 is the latest version of file *s.abc* below release 9. Similarly, omission of the sequence number, as in:

```
get -r4.3.2 s.abc
```

results in the retrieval of the branch delta with the highest sequence number on the given branch if it exists. (If the given branch does not exist, an error message results.) This might result in the following output:

```
4.3.2.8
89 lines
```

The `-t` keyletter is used to retrieve the latest (top) version in a particular release (i.e., when no `-r` keyletter is supplied or when its value is simply a release number). The latest version is defined as that delta which was produced most recently, independent of its location on the SCCS file tree. Thus, if the most recent delta in release 3 is 3.5,

```
get -r3 -t s.abc
```

might produce:

```
3.5
59 lines
```



However, if branch delta 3.2.1.5 were the latest delta (created after delta 3.5), the same command might produce:

3.2.1.5  
46 lines

#### Retrieval With Intent to Make a Delta

Specification of the `-e` keyletter to the `get(1)` command is an indication of the intent to make a delta, and as such, its use is restricted. The presence of this keyletter causes `get` to check:

1. The user list (a list of login names and/or group IDs of users allowed to make deltas) to determine if the login name or group ID of the user executing `get` is on that list. Note that a null (empty) user list behaves as if it contained all possible login names.
2. The release (R) of the version being retrieved satisfies the relation:

$$\text{floor} \leq R \leq \text{ceiling}$$

to determine if the release being accessed is a protected release. The "floor" and "ceiling" are specified as flags in the SCCS file.

3. The release (R) is not locked against editing. The "lock" is specified as a flag in the SCCS file.
4. Whether or not multiple concurrent edits are allowed for the SCCS file as specified by the `j` flag in the SCCS file.

A failure of any of the first three conditions causes the processing of the corresponding SCCS file to terminate.

If the above checks succeed, the `-e` keyletter causes the creation of a g-file in the current directory with mode 644 (readable by everyone, writable only by the owner) owned by the real user. If a writable g-file already exists, `get` terminates with an error. This is to prevent inadvertent destruction of a g-file that already exists and is being edited for the purpose of making a delta.

Any ID keywords appearing in the g-file are not substituted by `get` when the `e` keyletter is specified because the generated g-file is to be subsequently used to create another delta, and replacement of ID keywords would cause them to be permanently changed within the SCCS file. In view of this, `get` does not need to check for the presence of ID keywords within the g-file, so the message

No id keywords (cm7)

is never output when `get` is invoked with the `-e` keyletter.

In addition, the `-e` keyletter causes the creation (or updating) of a p-file which is used to pass information to the `delta(1)` command.

The following is an example of the use of the `-e` keyletter:

```
get -e s.abc
```

which produces (for example) on the standard output:

```
1.3  
new delta 1.4  
67 lines
```



If the `-r` and/or `-t` keyletters are used together with the `-e` keyletter, the version retrieved for editing is as specified by the `-r` and/or `-t` keyletters.

The keyletters `-i` and `-x` may be used to specify a list [see `get(1)` in the UNIX System User's Manual for the syntax of such a list] of deltas to be included and excluded, respectively, by `get`. Including a delta means forcing the changes that constitute the particular delta to be included in the retrieved version. This is useful if one wants to apply the same changes to more than one version of the SCCS file. Excluding a delta means forcing it to be not applied. This may be used to undo in the version of the SCCS file to be created the effects of a previous delta. Whenever deltas are included or excluded, `get` checks for possible interference between such deltas and those deltas that are normally used in retrieving the particular version of the SCCS file. Two deltas can interfere, for example, when each one changes the same line of the retrieved g-file. Any interference is indicated by a warning that shows the range of lines within the retrieved g-file in which the problem may exist. The user is expected to examine the g-file to determine whether a problem actually exists and to take whatever corrective measures (if any) are deemed necessary (e.g., edit the file).

**Warning:** *The `-i` and `-x` keyletters should be used with extreme care.*

The `-k` keyletter is provided to facilitate regeneration of a g-file that may have been accidentally removed or ruined subsequent to the execution of `get` with the `-e` keyletter or to simply generate a g-file in which the replacement of ID keywords has been suppressed. Thus, a g-file generated by the `-k` keyletter is identical to one produced by `get` executed with the `-e` keyletter. However, no processing related to the p-file takes place.

#### Concurrent Edits of Different SIDs

The ability to retrieve different versions of an SCCS file allows a number of deltas to be "in progress" at any given time. This means that a number of `get` commands with the `-e` keyletter may be executed on the same file provided that no two executions retrieve the same version (unless multiple concurrent edits are allowed).

The p-file (created by the `get` command invoked with the `-e` keyletter) is named by replacing the "s." in the SCCS file name with "p.". It is created in the directory containing the SCCS file, given mode 644 (readable by everyone, writable only by the owner), and owned by the effective user. The p-file contains the following information for each delta that is still "in progress":

- The SID of the retrieved version.
- The SID that will be given to the new delta when it is created.
- The login name of the real user executing `get`.

The first execution of `get -e` causes the creation of the p-file for the corresponding SCCS file. Subsequent executions only update the p-file with a line containing the above information. Before updating, however, `get` checks that no entry already in the p-file specifies as already retrieved the SID of the version to be retrieved unless multiple concurrent edits are allowed.

If both checks succeed, the user is informed that other deltas are in progress and processing continues. If either check fails, an error message results. It is important to note that the various executions of `get` should be carried out from different directories. Otherwise, only the first execution will succeed since subsequent executions would attempt to overwrite a writable g-file, which is an SCCS error condition. In practice, such multiple executions are performed by different users so that this problem does not arise since each user normally has



a different working directory. See "Protection" under the part "SCCS FILES" for a discussion of how different users are permitted to use SCCS commands on the same files.

Table 7.A shows, for the most useful cases, the version of an SCCS file retrieved by **get**, as well as the SID of the version to be eventually created by **delta**, as a function of the SID specified to **get**.

#### Concurrent Edits of Same SID

Under normal conditions, **gets** for editing (**-e** keyletter is specified) based on the same SID are not permitted to occur concurrently. That is, **delta(1)** must be executed before a subsequent **get** for editing is executed at the same SID as the previous **get**. However, multiple concurrent edits (defined to be two or more successive executions of **get** for editing based on the same retrieved SID) are allowed if the **j** flag is set in the SCCS file. Thus:

```
get -e s.abc
1.1
new delta 1.2
5 lines
```

may be immediately followed by:

```
get -e s.abc
1.1
new delta 1.1.1.1
5 lines
```

without an intervening execution of **delta**. In this case, a **delta** command corresponding to the first **get** produces delta 1.2 (assuming 1.1 is the latest [most recent] trunk delta), and the **delta** command corresponding to the second **get** produces delta 1.1.1.1.

#### Keyletters That Affect Output

Specification of the **-p** keyletter causes **get** to write the retrieved text to the standard output rather than to a **g-file**. In addition, all output normally directed to the standard output (such as the SID of the version retrieved and the number of lines retrieved) is directed instead to the diagnostic output. This may be used, for example, to create **g-files** with arbitrary names:

```
get -p s.abc > arbitrary-file-name
```

The **-p** keyletter is particularly useful when used with the **"!"** or **"\$"** arguments of the **send(1C)** command. For example:

```
send MOD=s.abc REL=3 compile
```

given that file *compile* contains:

```
//plicomp job job-card-information

//step1 exec plicc
//pli.sysin dd *
~ -s
~!get -p -rREL MOD
/*
//
```



will **send** the highest level of release 3 of file *s.abc*. Note that the line "**-s**", which causes **send** to make ID keyword substitutions before detecting and interpreting control lines, is necessary if **send** is to substitute "*s.abc*" for MOD and "3" for REL in the line "**~!get -p -rREL MOD**".

The **-s** keyletter suppresses all output that is normally directed to the standard output. Thus, the SID of the retrieved version, the number of lines retrieved, etc., are not output. This does not, however, affect messages to the diagnostic output. This keyletter is used to prevent nondiagnostic messages from appearing on the user's terminal and is often used in conjunction with the **-p** keyletter to "pipe" the output of **get**, as in:

```
get -p -s s.abc | nroff
```

The **-g** keyletter is supplied to suppress the actual retrieval of the text of a version of the SCCS file. This may be useful in a number of ways. For example, to verify the existence of a particular SID in an SCCS file, one may execute:

```
get -g -r4.3 s.abc
```

This outputs the given SID if it exists in the SCCS file or it generates an error message if it does not. Another use of the **-g** keyletter is in regenerating a p-file that may have been accidentally destroyed:

```
get -e -g s.abc
```

The **-l** keyletter causes the creation of an l-file, which is named by replacing the "s." of the SCCS file name with "l.". This file is created in the current directory with mode 444 (read-only) and is owned by the real user. It contains a table (whose format is described in **get** in the UNIX System User's Manual showing which deltas were used in constructing a particular version of the SCCS file. For example:

```
get -r2.3 -l s.abc
```

generates an l-file showing the deltas applied to retrieve version 2.3 of the SCCS file. Specifying a value of "p" with the **-l** keyletter, as in:

```
get -lp -r2.3 s.abc
```

causes the generated output to be written to the standard output rather than to the l-file. The **-g** keyletter may be used with the **-l** keyletter to suppress the actual retrieval of the text.

The **-m** keyletter is of use in identifying, line by line, the changes applied to an SCCS file. Specification of this keyletter causes each line of the generated g-file to be preceded by the SID of the delta that caused that line to be inserted. The SID is separated from the text of the line by a tab character.

The **-n** keyletter causes each line of the generated g-file to be preceded by the value of the **%M%** ID keyword and a tab character. The **-n** keyletter is most often used in a pipeline with **grep(1)**. For example, to find all lines that match a given pattern in the latest version of each SCCS file in a directory, the following may be executed:

```
get -p -n -s directory | grep pattern
```

If both the **-m** and **-n** keyletters are specified, each line of the generated g-file is preceded by the value of the **%M%** ID keyword and a tab (this is the effect of the **-n** keyletter) and followed by the line in the format produced by the **-m** keyletter. Because use of the **-m** keyletter and/or the **-n** keyletter causes the contents of the g-file to be modified, such a g-file must not be used for creating a delta. Therefore, neither the **-m** keyletter nor the **-n** keyletter may be specified together with the **-e** keyletter.

See **get(1)** in the UNIX System User's Manual for a full description of additional **get** keyletters.



## B. The "delta" Command

The **delta**(1) command is used to incorporate the changes made to a g-file into the corresponding SCCS file, i.e., to create a delta, and therefore, a new version of the file.

Invocation of the **delta** command requires the existence of a p-file. The **delta** command examines the p-file to verify the presence of an entry containing the user's login name. If none is found, an error message results. The **delta** command also performs the same permission checks that **get**(1) performs when invoked with the **-e** keyletter. If all checks are successful, **delta** determines what has been changed in the g-file by comparing it via **diff** with its own temporary copy of the g-file as it was before editing. This temporary copy of the g-file is called the d-file (its name is formed by replacing the "s." of the SCCS file name with "d.") and is obtained by performing an internal **get** at the SID specified in the p-file entry.

The required p-file entry is the one containing the login name of the user executing **delta** because the user who retrieved the g-file must be the one who will create the delta. However, if the login name of the user appears in more than one entry (i.e., the same user executed **get** with the **-e** keyletter more than once on the same SCCS file), the **-r** keyletter must be used with **delta** to specify an SID that uniquely identifies the p-file entry. This entry is the one used to obtain the SID of the delta to be created.

In practice, the most common invocation of **delta** is

```
delta s.abc
```

which prompts on the standard output (but only if it is a terminal):

```
comments?
```

to which the user replies with a description of why the delta is being made, terminating the reply with a new line character. The user's response may be up to 512 characters long with new lines, not intended to terminate the response, escaped by "\".

If the SCCS file has a **v** flag, **delta** first prompts with

```
MRs?
```

on the standard output. (Again, this prompt is printed only if the standard output is a terminal.) The standard input is then read for Modification Request (MR) numbers, separated by blanks and/or tabs, terminated in the same manner as the response to the prompt "comments?". In a tightly controlled environment, it is expected that deltas are created only as a result of some trouble report, change request, trouble ticket, etc. (collectively called here MRs) and that it is desirable or necessary to record such MR number(s) within each delta.

The **-y** and/or **-m** keyletters may be used to supply the commentary (comments and MR numbers, respectively) on the command line rather than through the standard input:

```
delta -y"descriptive comment" -m"mrnum1 mrnum2" s.abc
```

In this case, the corresponding prompts are not printed, and the standard input is not read. The **-m** keyletter is allowed only if the SCCS file has a **v** flag. These keyletters are useful when **delta** is executed from within a shell procedure. See **sh**(1) in the UNIX System User's Manual.

The commentary (comments and/or MR numbers), whether solicited by **delta** or supplied via keyletters, is recorded as part of the entry for the delta being created and applies to all SCCS files processed by the same invocation of **delta**. This implies that if **delta** is invoked with more than one file argument and the first file



named has a **v** flag all files named must have this flag. Similarly, if the first file named does not have this flag, then none of the files named may have it. Any file that does not conform to these rules is not processed.

When processing is complete, **delta** outputs (on the standard output) the SID of the created delta (obtained from the p-file entry) and the counts of lines inserted, deleted, and left unchanged by the delta. Thus, a typical output might be:

```
1.4
14 inserted
7 deleted
345 unchanged
```

It is possible that the counts of lines reported as inserted, deleted, or unchanged by **delta** do not agree with the user's perception of the changes applied to the g-file. The reason for this is that there usually are a number of ways to describe a set of such changes, especially if lines are moved around in the g-file, and **delta** is likely to find a description that differs from the user's perception. However, the total number of lines of the new delta (the number inserted plus the number left unchanged) should agree with the number of lines in the edited g-file.

If in the process of making a delta **delta** finds no ID keywords in the edited g-file, the message

```
No id keywords (cm7)
```

is issued after the prompts for commentary but before any other output. This indicates that any ID keywords that may have existed in the SCCS file have been replaced by their values or deleted during the editing process. This could be caused by creating a delta from a g-file that was created by a **get** without the **-e** keyletter (recall that ID keywords are replaced by **get** in that case) or by accidentally deleting or changing the ID keywords during the editing of the g-file. Another possibility is that the file may never have had any ID keywords. In any case, it is left up to the user to determine what remedial action is necessary, but the delta is made, unless there is an **i** flag in the SCCS file indicating that this should be treated as a fatal error. In this last case, the delta is not created.

After processing of an SCCS file is complete, the corresponding p-file entry is removed from the p-file. All updates to the p-file are made to a temporary copy, the q-file, whose use is similar to the use of the x-file which is described in the part "SCCS COMMAND CONVENTIONS". If there is only one entry in the p-file, then the p-file itself is removed.

In addition, **delta** removes the edited g-file unless the **-n** keyletter is specified. Thus:

```
delta -n s.abc
```

will keep the g-file upon completion of processing.

The **-s** (silent) keyletter suppresses all output that is normally directed to the standard output, other than the prompts "comments?" and "MRs?". Thus, use of the **-s** keyletter together with the **-y** keyletter (and possibly, the **-m** keyletter) causes **delta** neither to read the standard input nor to write the standard output.

The differences between the g-file and the d-file (see above), which constitute the delta, may be printed on the standard output by using the **-p** keyletter. The format of this output is similar to that produced by **diff**.

#### C. The "admin" Command

The **admin(1)** command is used to administer SCCS files, that is, to create new SCCS files and to change parameters of existing ones. When an SCCS file is created, its parameters are initialized by use of keyletters



or are assigned default values if no keyletters are supplied. The same keyletters are used to change the parameters of existing files.

Two keyletters are supplied for use in conjunction with detecting and correcting "corrupted" SCCS files. (Discussed in "Auditing" under the part "SCCS FILES".) Newly created SCCS files are given mode 444 (read-only) and are owned by the effective user. Only a user with write permission in the directory containing the SCCS file may use the **admin** command upon that file.

#### Creation of SCCS Files

An SCCS file may be created by executing the command

```
admin -ifirst s.abc
```

in which the value "first" of the **-i** keyletter specifies the name of a file from which the text of the initial delta of the SCCS file *s.abc* is to be taken. Omission of the value of the **-i** keyletter indicates that **admin** is to read the standard input for the text of the initial delta. Thus, the command

```
admin -i s.abc < first
```

is equivalent to the previous example. If the text of the initial delta does not contain ID keywords, the message

```
No id keywords (cm7)
```

is issued by **admin** as a warning. However, if the same invocation of the command also sets the **i** flag (not to be confused with the **-i** keyletter), the message is treated as an error and the SCCS file is not created. Only one SCCS file may be created at a time using the **-i** keyletter.

When an SCCS file is created, the release number assigned to its first delta is normally "1", and its level number is always "1". Thus, the first delta of an SCCS file is normally "1.1". The **-r** keyletter is used to specify the release number to be assigned to the first delta. Thus:

```
admin -ifirst -r3 s.abc
```

indicates that the first delta should be named "3.1" rather than "1.1". Because this keyletter is only meaningful in creating the first delta, its use is only permitted with the **-i** keyletter.

#### Inserting Commentary for the Initial Delta

When an SCCS file is created, the user may choose to supply commentary stating the reason for creation of the file. This is done by supplying comments (**-y** keyletter) and/or MR numbers (**-m** keyletter) in exactly the same manner as for **delta**. The creation of an SCCS file may sometimes be the direct result of an MR. If comments (**-y** keyletter) are omitted, a comment line of the form

```
date and time created YY/MM/DD HH:MM:SS by logname
```

is automatically generated.

If it is desired to supply MR numbers (**-m** keyletter), the **v** flag must also be set (using the **-f** keyletter described below). The **v** flag simply determines whether or not MR numbers must be supplied when using any SCCS command that modifies a "delta commentary" [see **scsfile(1)** in the UNIX System User's Manual] in the SCCS file. Thus:

```
admin -ifirst -mmrnum1 -fv s.abc
```



Note that the `-y` and `-m` keyletters are only effective if a new SCCS file is being created.

#### Initialization and Modification of SCCS File Parameters

The portion of the SCCS file reserved for descriptive text may be initialized or changed through the use of the `-t` keyletter. The descriptive text is intended as a summary of the contents and purpose of the SCCS file.

When an SCCS file is being created and the `-t` keyletter is supplied, it must be followed by the name of a file from which the descriptive text is to be taken. For example, the command

```
admin -ifirst -tdesc s.abc
```

specifies that the descriptive text is to be taken from file `desc`.

When processing an existing SCCS file, the `-t` keyletter specifies that the descriptive text (if any) currently in the file is to be replaced with the text in the named file. Thus:

```
admin -tdesc s.abc
```

specifies that the descriptive text of the SCCS file is to be replaced by the contents of `desc`; omission of the file name after the `-t` keyletter as in

```
admin -t s.abc
```

causes the removal of the descriptive text from the SCCS file.

The flags of an SCCS file may be initialized, changed, or deleted through the use of the `-f` and `-d` keyletters, respectively. The flags of an SCCS file are used to direct certain actions of the various commands. See `admin(1)` in the UNIX System User's Manual for a description of all the flags. For example, the `i` flag specifies that the warning message stating there are no ID keywords contained in the SCCS file should be treated as an error, and the `d` (default SID) flag specifies the default version of the SCCS file to be retrieved by the `get` command. The `-f` keyletter is used to set a flag and, possibly, to set its value. For example:

```
admin -ifirst -fi -fmmodname s.abc
```

sets the `i` flag and the `m` (module name) flag. The value "modname" specified for the `m` flag is the value that the `get` command will use to replace the `%M%` ID keyword. (In the absence of the `m` flag, the name of the g-file is used as the replacement for the `%M%` ID keyword.) Note that several `-f` keyletters may be supplied on a single invocation of `admin` and that `-f` keyletters may be supplied whether the command is creating a new SCCS file or processing an existing one.

The `-d` keyletter is used to delete a flag from an SCCS file and may only be specified when processing an existing file. As an example, the command

```
admin -dm s.abc
```

removes the `m` flag from the SCCS file. Several `-d` keyletters may be supplied on a single invocation of `admin` and may be intermixed with `-f` keyletters.

The SCCS files contain a list (user list) of login names and/or group IDs of users who are allowed to create deltas. This list is empty by default which implies that anyone may create deltas. To add login names and/or group IDs to the list, the `-a` keyletter is used. For example:

```
admin -axyz -awql -a1234 s.abc
```



adds the login names "xyz" and "wql" and the group ID "1234" to the list. The `-a` keyletter may be used whether `admin` is creating a new SCCS file or processing an existing one and may appear several times. The `-e` keyletter is used in an analogous manner if one wishes to remove (erase) login names or group IDs from the list.

#### D. The "prs" Command

The `prs(1)` command is used to print on the standard output all or parts of an SCCS file in a format, called the output "data specification", supplied by the user via the `-d` keyletter. The data specification is a string consisting of SCCS file data keywords (not to be confused with `get` ID keywords) interspersed with optional user text.

Data keywords are replaced by appropriate values according to their definitions. For example:

`:I:`

is defined as the data keyword that is replaced by the SID of a specified delta. Similarly, `:F:` is defined as the data keyword for the SCCS file name currently being processed, and `:C:` is defined as the comment line associated with a specified delta. All parts of an SCCS file have an associated data keyword. For a complete list of the data keywords, see `prs(1)` in the UNIX System User's Manual.

There is no limit to the number of times a data keyword may appear in a data specification. Thus, for example:

```
prs -d":I: this is the top delta for :F: :I:" s.abc
```

may produce on the standard output

```
2.1 this is the top delta for s.abc 2.1
```

Information may be obtained from a single delta by specifying the SID of that delta using the `-r` keyletter. For example:

```
prs -d":F:: :I: comment line is: :C:" -r1.4 s.abc
```

may produce the following output:

```
s.abc: 1.4 comment line is: THIS IS A COMMENT
```

If the `-r` keyletter is not specified, the value of the SID defaults to the most recently created delta.

In addition, information from a range of deltas may be obtained by specifying the `-l` or `-e` keyletters. The `-e` keyletter substitutes data keywords for the SID designated via the `-r` keyletter and all deltas created earlier. The `-l` keyletter substitutes data keywords for the SID designated via the `-r` keyletter and all deltas created later. Thus, the command

```
prs -d:I: -r1.4 -e s.abc
```

may output

```
1.4
1.3
1.2.1.1
1.2
1.1
```



and the command

```
prs -d:I: -r1.4 -l s.abc
```

may produce

```
3.3
3.2
3.1
2.2.1.1
2.2
2.1
1.4
```

Substitution of data keywords for all deltas of the SCCS file may be obtained by specifying both the `-e` and `-l` keyletters.

#### E. The "help" Command

The `help(1)` command prints explanations of SCCS commands and of messages that these commands may print. Arguments to `help`, zero or more of which may be supplied, are simply the names of SCCS commands or the code numbers that appear in parentheses after SCCS messages. If no argument is given, `help` prompts for one. The `help` command has no concept of keyletter arguments or file arguments. Explanatory information related to an argument, if it exists, is printed on the standard output. If no information is found, an error message is printed. Note that each argument is processed independently, and an error resulting from one argument will not terminate the processing of the other arguments.

Explanatory information related to a command is a synopsis of the command. For example:

```
help ge5 rmdel
```

produces

```
ge5:
"nonexistent sid"
The specified sid does not exist in the
given file.
Check for typographical errors
```

```
rmdel:
rmdel -rSID name ...
```

#### F. The "rmdel" Command

The `rmdel(1)` command is provided to allow removal of a delta from an SCCS file. Its use should be reserved for those cases in which incorrect global changes were made a part of the delta to be removed.

The delta to be removed must be a "leaf" delta. That is, it must be the latest (most recently created) delta on its branch or on the trunk of the SCCS file tree. In Fig. 7.3, only deltas 1.3.1.2, 1.3.2.2, and 2.2 can be removed; once they are removed then deltas 1.3.2.1 and 2.1 can be removed, etc.

To be allowed to remove a delta, the effective user must have write permission in the directory containing the SCCS file. In addition, the real user must either be the one who created the delta being removed or be the owner of the SCCS file and its directory.



The `-r` keyletter, which is mandatory, is used to specify the complete SID of the delta to be removed (i.e., it must have two components for a trunk delta and four components for a branch delta). Thus:

```
rmidel -r2.3 s.abc
```

specifies the removal of (trunk) delta "2.3" from the SCCS file. Before removal of the delta, **rmidel** checks that the release number (R) of the given SID satisfies the relation:

$$\text{floor} \leq R \leq \text{ceiling}$$

The **rmidel** command also checks that the SID specified is not that of a version for which a **get** for editing has been executed and whose associated **delta** has not yet been made. In addition, the login name or group ID of the user must appear in the file's "user list", or the "user list" must be empty. Also, the release specified can not be locked against editing (i.e., if the **l** flag is set [see **admin(1)** in the *UNIX System User's Manual*], the release specified must not be contained in the list). If these conditions are not satisfied, processing is terminated, and the delta is not removed. After the specified delta has been removed, its type indicator in the "delta table" of the SCCS file is changed from "D" (delta) to "R" (removed).

#### G. The "cdc" Command

The **cdc** command is used to change a delta's commentary that was supplied when that delta was created. Its invocation is analogous to that of the **rmidel** command, except that the delta to be processed is not required to be a leaf delta. For example:

```
cdc -r3.4 s.abc
```

specifies that the commentary of delta "3.4" of the SCCS file is to be changed.

The new commentary is solicited by **cdc** in the same manner as that of **delta**. The old commentary associated with the specified delta is kept, but it is preceded by a comment line indicating that it has been changed (i.e., superseded), and the new commentary is entered ahead of this comment line. The "inserted" comment line records the login name of the user executing **cdc** and the time of its execution.

The **cdc** command also allows for the deletion of selected MR numbers associated with the specified delta. This is specified by preceding the selected MR numbers by the character "!". Thus:

```
cdc -r1.4 s.abc
MRs? mrnum3 !mrnum1
comments? deleted wrong MR number and inserted correct MR number
```

inserts "mrnum3" and deletes "mrnum1" for delta 1.4.

#### H. The "what" Command

The **what(1)** command is used to find identifying information within any UNIX operating system file whose name is given as an argument to **what**. Directory names and a name of "-" (a lone minus sign) are not treated specially, as they are by other SCCS commands, and no keyletters are accepted by the command.

The **what** command searches the given file(s) for all occurrences of the string "@(#)", which is the replacement for the %Z% ID keyword (see **get**), and prints (on the standard output) what follows that string until the first double quote ("), greater than (>), backslash (\), new line, or (nonprinting) NUL character. Thus, for example, if the SCCS file *s.prog.c* (a C language program), contains the following line:

```
char id[] "%Z% %M%:%I%";
```



and then the command

```
get -r3.4 s.prog.c
```

is executed, the resulting g-file is compiled to produce "prog.o" and "a.out". Then the command

```
what prog.c prog.o a.out
```

produces

```
prog.c:
  prog.c:3.4
prog.o:
  prog.c:3.4
a.out:
  prog.c:3.4
```

The string searched for by **what** need not be inserted via an ID keyword of **get**; it may be inserted in any convenient manner.

#### I. The "sccsdiff" Command

The **sccsdiff(1)** command determines (and prints on the standard output) the differences between two specified versions of one or more SCCS files. The versions to be compared are specified by using the **-r** keyletter, whose format is the same as for the **get** command. The two versions must be specified as the first two arguments to this command in the order they were created, i.e., the older version is specified first. Any following keyletters are interpreted as arguments to the **pr** command (which actually prints the differences) and must appear before any file names. The SCCS files to be processed are named last. Directory names and a name of "-" (a lone minus sign) are not acceptable to **sccsdiff**.

The differences are printed in the form generated by **diff(1)**. The following is an example of the invocation of **sccsdiff**:

```
sccsdiff -r3.4 -r5.6 s.abc
```

#### J. The "comb" Command

The **comb** command generates a "shell procedure" [see **sh(1)** in the UNIX System User's Manual] which attempts to reconstruct the named SCCS files so that the reconstructed files are smaller than the originals. The generated shell procedure is written on the standard output. Named SCCS files are reconstructed by discarding unwanted deltas and combining specified other deltas. The SCCS files that contain deltas that are no longer useful are to be discarded. It is not recommended that **comb(1)** be used as a matter of routine; its use should be restricted to a very small number of times in the life of an SCCS file.

In the absence of any keyletters, **comb** preserves only leaf deltas and the minimum number of ancestor deltas necessary to preserve the "shape" of the SCCS file tree. The effect of this is to eliminate middle deltas on the trunk and on all branches of the tree. Thus, in Fig. 7.3, deltas 1.2, 1.3.2.1, 1.4, and 2.1 would be eliminated. Some of the keyletters are summarized as follows:

The **-p** keyletter specifies the oldest delta that is to be preserved in the reconstruction. All older deltas are discarded.

The **-c** keyletter specifies a list [see **get(1)** in the UNIX System User's Manual for the syntax of such a list] of deltas to be preserved. All other deltas are discarded.



The **-s** keyletter causes the generation of a shell procedure, which when run, produces only a report summarizing the percentage space (if any) to be saved by reconstructing each named SCCS file. It is recommended that **comb** be run with this keyletter (in addition to any others desired) before any actual reconstructions.

It should be noted that the shell procedure generated by **comb** is not guaranteed to save space. In fact, it is possible for the reconstructed file to be larger than the original. Note, too, that the shape of the SCCS file tree may be altered by the reconstruction process.

#### K. The "val" Command

The **val(1)** command is used to determine if a file is an SCCS file meeting the characteristics specified by an optional list of keyletter arguments. Any characteristics not met are considered errors.

The **val** command checks for the existence of a particular delta when the SID for that delta is explicitly specified via the **-r** keyletter. The string following the **-y** or **-m** keyletter is used to check the value set by the **t** or **m** flag, respectively [see **admin(1)** in the UNIX System User's Manual for a description of the flags]

The **val** command treats the special argument **"-"** differently from other SCCS commands. This argument allows **val** to read the argument list from the standard input as opposed to obtaining it from the command line. The standard input is read until end of file. This capability allows for one invocation of **val** with different values for the keyletter and file arguments. For example:

```
val -  
-yc -mabc s.abc  
-mxyz -ypl1 s.xyz
```

first checks if file *s.abc* has a value **"c"** for its **"type"** flag and value **"abc"** for the **"module name"** flag. Once processing of the first file is completed, **val** then processes the remaining files, in this case, *s.xyz*, to determine if they meet the characteristics specified by the keyletter arguments associated with them.

The **val** command returns an 8-bit code; each bit set indicates the occurrence of a specific error (see **val** for a description of the possible errors and their codes). In addition, an appropriate diagnostic is printed unless suppressed by the **-s** keyletter. A return code of **"0"** indicates all named files met the characteristics specified.

### SCCS FILES

This part discusses several topics that must be considered before extensive use is made of SCCS. These topics deal with the protection mechanisms relied upon by SCCS, the format of SCCS files, and the recommended procedures for auditing SCCS files.

#### A. Protecting

The SCCS relies on the capabilities of the UNIX operating system for most of the protection mechanisms required to prevent unauthorized changes to SCCS files (i.e., changes made by non-SCCS commands). The only protection features provided directly by SCCS are the **"release lock"** flag, the **"release floor"** and **"ceiling"** flags, and the **"user list"**.

New SCCS files created by the **admin** command are given mode 444 (read-only). It is recommended that this mode not be changed as it prevents any direct modification of the files by non-SCCS commands. It is further recommended that the directories containing SCCS files be given mode 755 which allows only the owner of the directory to modify its contents.

The SCCS files should be kept in directories that contain only SCCS files and any temporary files created by SCCS commands. This simplifies protection and auditing of SCCS files. The contents of directories should correspond to convenient logical groupings, e.g., subsystems of a large project.



The SCCS files must have only one link (name) because the commands that modify SCCS files do so by creating a copy of the file (the x-file, see "SCCS COMMAND CONVENTIONS") and, upon completion of processing, remove the old file and rename the x-file. If the old file has more than one link, this would break such additional links. Rather than process such files, SCCS commands produce an error message. All SCCS files must have names that begin with "s."

When only one user uses SCCS, the real and effective user IDs are the same; and the user ID owns the directories containing SCCS files. Therefore, SCCS may be used directly without any preliminary preparation.

However, in those situations in which several users with unique user IDs are assigned responsibility for one SCCS file (for example, in large software development projects), one user (equivalently, one user ID) must be chosen as the "owner" of the SCCS files and be the one who will "administer" them. This user is termed the "SCCS administrator" for that project. Because other users of SCCS do not have the same privileges and permissions as the SCCS administrator, they are not able to execute directly those commands that require write permission in the directory containing the SCCS files. Therefore, a project-dependent program is required to provide an interface to the `get(1)`, `delta(1)`, and if desired, `rmDEL(1)`, and `cdc(1)` commands.

The interface program must be owned by the SCCS administrator and must have the "set user ID on execution" bit "on" [see `chmod(1)` in the *UNIX System User's Manual*], so that the effective user ID is the user ID of the administrator. This program invokes the desired SCCS command and causes it to inherit the privileges of the interface program for the duration of that command's execution. Thus, the owner of an SCCS file can modify it at will. Other users whose login names or group IDs are in the "user list" for that file (but who are not its owner) are given the necessary permissions only for the duration of the execution of the interface program. These other users are thus able to modify the SCCS files only through the use of `delta` and, possibly, `rmDEL` and `cdc`. The project-dependent interface program, as its name implies, must be custom-built for each project.

## B. Formatting

The SCCS files are composed of lines of ASCII text arranged in six parts as follows:

|                  |                                                                                                                     |
|------------------|---------------------------------------------------------------------------------------------------------------------|
| Checksum         | A line containing the "logical" sum of all the characters of the file ( <i>not</i> including this checksum itself). |
| Delta Table      | Information about each delta, such as type, SID, date and time of creation, and commentary.                         |
| User Names       | List of login names and/or group IDs of users who are allowed to modify the file by adding or removing deltas.      |
| Flags            | Indicators that control certain actions of various SCCS commands.                                                   |
| Descriptive Text | Arbitrary text provided by the user; usually a summary of the contents and purpose of the file.                     |
| Body             | Actual text that is being administered by SCCS, intermixed with internal SCCS control lines.                        |

Detailed information about the contents of the various sections of the file may be found in `sccsfile`. The checksum is the only portion of the file which is of interest below.

It is important to note that because SCCS files are ASCII files, they may be processed by various UNIX operating system commands, such as `ed(1)`, `grep(1)`, and `cat(1)`. This is very convenient in those instances in which



an SCCS file must be modified manually (e.g., when the time and date of a delta was recorded incorrectly because the system clock was set incorrectly) or when it is desired to simply look at the file.

**Caution:** *Extreme care should be exercised when modifying SCCS files with non-SCCS commands.*

### C. Auditing

On rare occasions, perhaps due to an operating system or hardware malfunction, an SCCS file or portions of it (i.e., one or more "blocks") can be destroyed. The SCCS commands (like most UNIX operating system commands) issue an error message when a file does not exist. In addition, SCCS commands use the checksum stored in the SCCS file to determine whether a file has been corrupted since it was last accessed (possibly by having lost one or more blocks or by having been modified with, for example, `ed`). No SCCS command will process a corrupted SCCS file except the `admin` command with the `-h` or `-z` keyletters, as described below.

It is recommended that SCCS files be audited (checked) for possible corruptions on a regular basis. The simplest and fastest way to perform an audit is to execute the `admin` command with the `-h` keyletter on all SCCS files:

```
admin -h s.file1 s.file2 ...  
      or  
admin -h directory1 directory2 ...
```

If the new checksum of any file is not equal to the checksum in the first line of that file, the message

```
corrupted file (co6)
```

is produced for that file. This process continues until all the files have been examined. When examining directories (as in the second example above), the process just described will not detect missing files. A simple way to detect whether any files are missing from a directory is to periodically execute the `ls(1)` command on that directory and compare the outputs of the most current and the previous executions. Any file whose name appears in the previous output but not in the current one has been removed by some means.

Whenever a file has been corrupted, the manner the file is restored depends upon the extent of the corruption. If damage is extensive, the best solution is to contact the local UNIX operating system operations group and request the file be restored from a backup copy. In the case of minor damage, repair through use of the editor `ed` may be possible. In the latter case after such repair, the following command must be executed:

```
admin -z s.file
```

The purpose of this is to recompute the checksum to bring it into agreement with the actual contents of the file. After this command is executed on a file, any corruption that existed in the file will no longer be detectable.

## AN SCCS INTERFACE PROGRAM

### A. General

In order to permit UNIX operating system users with different user identification numbers (user IDs) to use SCCS commands upon the same files, an SCCS interface program is provided to temporarily grant the necessary file access permissions to these users. This part discusses the creation and use of such an interface program. The SCCS interface program may also be used as a preprocessor to SCCS commands since it can perform operations upon its arguments.



**B. Function**

When only one user uses SCCS, the real and effective user IDs are the same; and that user's ID owns the directories containing SCCS files. However, there are situations (e.g., in large software development projects) in which it is practical to allow more than one user to make changes to the same set of SCCS files. In these cases, one user must be chosen as the "owner" of the SCCS files and be the one who will "administer" them (e.g., by using the `admin(1)` command). This user is termed the "SCCS administrator" for that project. Since other users of SCCS do not have the same privileges and permissions as the SCCS administrator, the other users are not able to execute directly those commands that require write permission in the directory containing the SCCS files. Therefore, a project-dependent program is required to provide an interface to the `get(1)`, `delta(1)`, and if desired, `rmidel(1)`, `cdc(1)`, and `unget(1)` commands. Other SCCS commands either do not require write permission in the directory containing SCCS files or are (generally) reserved for use only by the administrator.

The interface program must be owned by the SCCS administrator, must be executable by nonowners, and must have the "set user ID on execution" bit "on" [see `chmod(1)` in the UNIX System User's Manual] so that, when executed, the effective user ID is the user ID of the administrator. This program's function is to invoke the desired SCCS command and to cause it to inherit the privileges of the SCCS administrator for the duration of that command's execution. In this manner, the owner of an SCCS file (the administrator) can modify it at will. Other users whose login names are in the user list for that file (but who are not its owners) are given the necessary permissions only for the duration of the execution of the interface program. They are thus able to modify the SCCS files only through the use of `delta`, and possibly, `rmidel`, and `cdc`.

**C. A Basic Program**

When a UNIX operating system program is executed, the program is passed as argument 0, which is the name that invoked the program, and followed by any additional user-supplied arguments. Thus, if a program is given a number of links (names), the program may alter its processing depending upon which link invokes the program. This mechanism is used by an SCCS interface program to determine which SCCS command it should subsequently invoke [see `exec(2)` in the UNIX System User's Manual].

A generic interface program (`inter.c`, written in C language) is shown in Table 7.B. Note the reference to the (unsupplied) function "filearg". This is intended to demonstrate that the interface program may also be used as a preprocessor to SCCS commands. For example, function "filearg" could be used to modify file arguments to be passed to the SCCS command by supplying the full pathname of a file, thus avoiding extraneous typing by the user. Also, the program could supply any additional (default) keyletter arguments desired.

**D. Linking and Use**

In general, the following demonstrates the steps to be performed by the SCCS administrator to create the SCCS interface program. It is assumed, for the purposes of the discussion, that the interface program `inter.c` resides in directory `"/x1/xyz/scs"`. Thus, the command sequence

```
cd /x1/xyz/scs
cc ... inter.c -o inter ...
```

compiles `inter.c` to produce the executable module `inter` (the "..." represent other arguments that may be required). The proper mode and the "set user ID on execution" bit are set by executing:

```
chmod 4755 inter
```



For example, new links are created by:

```
ln inter get
ln inter delta
ln inter rmdel
```

The names of the links may be arbitrary provided the interface program is able to determine from them the names of SCCS commands to be invoked. Subsequently, any user whose shell parameter *PATH* [see *sh(1)* in the *UNIX System User's Manual*] specifies directory *"/x1/xyz/scs"* as the one to be searched first for executable commands may execute, for example:

```
get -e /x1/xyz/scs/s.abc
```

from any directory to invoke the interface program (via its link "get"). The interface program then executes *"/usr/bin/get"* (the actual SCCS *get* command) upon the named file. As previously mentioned, the interface program could be used to supply the pathname *"/x1/xyz/scs"* so that the user would only have to specify

```
get -e s.abc
```

to achieve the same results.



TABLE 7.A

## DETERMINATION OF NEW SID

| CASE | SID SPECIFIED* | -b KEYLETTER USED† | OTHER CONDITIONS                            | SID RETRIEVED | SID OF DELTA TO BE CREATED |
|------|----------------|--------------------|---------------------------------------------|---------------|----------------------------|
| 1    | none‡          | no                 | R defaults to mR                            | mR.mL         | mR.(mL + 1)                |
| 2    | none‡          | yes                | R defaults to mR                            | mR.mL         | mR.mL.(mB + 1).1           |
| 3    | R              | no                 | R > mR                                      | mR.mL         | R.1§                       |
| 4    | R              | no                 | R = mR                                      | mR.mL         | mR.(mL + 1)                |
| 5    | R              | yes                | R > mR                                      | mR.mL         | mR.mL.(mB + 1).1           |
| 6    | R              | yes                | R = mR                                      | mR.mL         | mR.mL.(mB + 1).1           |
| 7    | R              | —                  | R < mR and R does not exist                 | hR.mL**       | hR.mL.(mB + 1).1           |
| 8    | R              | —                  | Trunk successor in release > R and R exists | R.mL          | R.mL.(mB + 1).1            |
| 9    | R.L            | no                 | No trunk successor                          | R.L           | R.(L + 1)                  |
| 10   | R.L            | yes                | No trunk successor                          | R.L           | R.L.(mB + 1).1             |
| 11   | R.L            | —                  | Trunk successor in release ≥ R              | R.L           | R.L.(mB + 1).1             |
| 12   | R.L.B          | no                 | No branch successor                         | R.L.B.mS      | R.L.B.(mS + 1)             |
| 13   | R.L.B          | yes                | No branch successor                         | R.L.B.mS      | R.L.(mB + 1).1             |
| 14   | R.L.B.S        | no                 | No branch successor                         | R.L.B.S       | R.L.B.(S + 1)              |
| 15   | R.L.B.S        | yes                | No branch successor                         | R.L.B.S       | R.L.(mB + 1).1             |
| 16   | R.L.B.S        | —                  | Branch successor                            | R.L.B.S       | R.L.(mB + 1).1             |

\* "R", "L", "B", and "S" are the "release", "level", "branch", and "sequence" components of the SID, respectively; "m" means "maximum". Thus, for example, "R.mL" means "the maximum level number within release R"; "R.L.(mB + 1).1" means "the first sequence number on the new branch (i.e., maximum branch number plus 1) of level L within release R". Note that if the SID specified is of the form "R.L", "R.L.B", or "R.L.B.S", each of the specified components must exist.

† The -b keyletter is effective only if the b flag [see admin(1)] is present in the file. In this table, an entry of "—" means "irrelevant".

‡ This case applies if the d (default SID) flag is not present in the file. If the d flag is present in the file, the SID obtained from the d flag is interrupted as if it had been specified on the command line. Thus, one of the other cases in this table applies.

§ This case is used to force the creation of the first delta in a new release.

\*\* "hR" is the highest existing release that is lower than the specified, nonexistent, release R.



TABLE 7.B

## SCCS INTERFACE PROGRAM "inter.c"

```
main(argc, argv)
int argc;
char *argv[];
{
    register int i;
    char cmdstr[LENGTH]

    /*
    Process file arguments (those that don't begin with "-").
    */
    for (i = 1; i < argc; i++)
        if (argv[i][0] != '-')
            argv[i] = filearg(argv[i]);

    /*
    Get "simple name" of name used to invoke this program
    (i.e., strip off directory-name prefix, if any).
    */
    argv[0] = sname(argv[0]);

    /*
    Invoke actual SCCS command, passing arguments.
    */
    sprintf(cmdstr, "/usr/bin/%s", argv[0]);
    execv(cmdstr, argv);
}
```



# Programming Guide

## UNIX System



This dokument was prepared with specific references to use of the UNIX system on a particular processer, the Western Electric 3B20S, which is not presently available except for internal use within the Bell System. However, the information contained herein is generally applicable to use of the UNIX system on various processors which are available in the general trade.

Trademarks:

|                             |                           |
|-----------------------------|---------------------------|
| MUNIX, CADMUS               | for PCS                   |
| UNIX                        | for Bell Laboratories     |
| DEC, PDP, VAX               | for DEC                   |
| MASSBUS, UNIBUS             |                           |
| KODAK, EKTAMATIC            | for Eastman Kodak Company |
| Mohrflow, Mohrdry,          | for Mohr Lino-Saw Comp.   |
| Mohrchem                    |                           |
| TEKTRONIX                   | for Tektronik, Inc.       |
| TELETYPE                    | for Teletype Corporation  |
| TRENDATA 4000A <sup>®</sup> | for Trendata Corporation  |
| Versatec                    | for Versatec Corporation  |
| DIABLO                      | for Xerox Corporation     |

Copyright 1984 by  
PCS GmbH, Pfälzer-Wald-Strasse 38, D-8000 München 90, tel. (089) 67804-0

The information contained herein is the property of PCS and shall neither be reproduced in whole or in part without PCS's prior written approval nor be implied to grant any license to make, use or sell equipment manufactured herewith.

PCS reserves the right to make changes without notice in the specifications and materials contained herein and shall not be responsible for any damages (including consequential) caused by reliance on the materials presented.



## PROGRAMMING GUIDE

## UNIX SYSTEM

| CONTENTS                              | PAGE |
|---------------------------------------|------|
| 1. INTRODUCTION . . . . .             | 9    |
| 2. AN INTRODUCTION TO SHELL . . . . . | 11   |
| INTRODUCTION . . . . .                | 11   |
| SIMPLE COMMANDS . . . . .             | 11   |
| A. Background Commands . . . . .      | 11   |
| B. Input/Output Redirection . . . . . | 12   |
| C. Pipelines and Filters . . . . .    | 12   |
| D. File Name Generation . . . . .     | 13   |
| E. Quoting . . . . .                  | 14   |
| F. Prompting by the Shell . . . . .   | 14   |
| G. The Shell and Login . . . . .      | 14   |
| H. Summary . . . . .                  | 15   |
| SHELL PROCEDURES . . . . .            | 15   |
| A. Control Flow—"for" . . . . .       | 16   |
| B. Control Flow—"case" . . . . .      | 17   |
| C. Here Documents . . . . .           | 18   |
| D. Shell Variables . . . . .          | 19   |
| E. The "test" Command . . . . .       | 21   |
| F. Control Flow—"while" . . . . .     | 21   |
| G. Control Flow—"if" . . . . .        | 22   |



| CONTENTS                                              | PAGE |
|-------------------------------------------------------|------|
| H. Debugging Shell Procedures . . . . .               | 24   |
| I. The "man" Command . . . . .                        | 25   |
| KEYWORD PARAMETERS . . . . .                          | 26   |
| A. Parameter Transmission . . . . .                   | 26   |
| B. Parameter Substitution . . . . .                   | 27   |
| C. Command Substitution . . . . .                     | 27   |
| D. Evaluation and Quoting . . . . .                   | 28   |
| E. Error Handling . . . . .                           | 30   |
| F. Fault Handling . . . . .                           | 31   |
| G. Command Execution . . . . .                        | 33   |
| H. Invoking the Shell . . . . .                       | 34   |
| THE SHELL TUTORIAL . . . . .                          | 37   |
| INTRODUCTION . . . . .                                | 37   |
| OVERVIEW OF THE UNIX SYSTEM ENVIRONMENT . . . . .     | 37   |
| A. File System . . . . .                              | 37   |
| B. UNIX System Processes . . . . .                    | 38   |
| SHELL BASICS . . . . .                                | 39   |
| A. Commands . . . . .                                 | 39   |
| B. How the Shell Finds Commands . . . . .             | 40   |
| C. Generation of Argument Lists . . . . .             | 40   |
| D. Shell Variables . . . . .                          | 41   |
| E. Quoting Mechanisms . . . . .                       | 45   |
| F. Redirection of Input and Output . . . . .          | 45   |
| G. Command Lines and Pipelines . . . . .              | 47   |
| H. Examples . . . . .                                 | 47   |
| I. Changing of the Shell and .profile State . . . . . | 48   |



| CONTENTS                                                     | PAGE |
|--------------------------------------------------------------|------|
| USING THE SHELL AS A COMMAND: SHELL PROCEDURES . . . . .     | 49   |
| A. A Command's Environment . . . . .                         | 49   |
| B. Invoking the Shell . . . . .                              | 49   |
| C. Passing Arguments to the Shell—"shift" . . . . .          | 50   |
| D. Control Commands . . . . .                                | 51   |
| E. Special Shell Commands . . . . .                          | 59   |
| F. Creation and Organization of Shell Procedures . . . . .   | 60   |
| G. More about Execution Flags . . . . .                      | 61   |
| MISCELLANEOUS SUPPORTING COMMANDS AND FEATURES . . . . .     | 61   |
| A. Conditional Evaluation—"test" . . . . .                   | 61   |
| B. Reading a Line—"line" . . . . .                           | 62   |
| C. Simple Output—"echo" . . . . .                            | 62   |
| D. Expression Evaluation—"expr" . . . . .                    | 63   |
| E. "true" and "false" . . . . .                              | 63   |
| F. Input/Output Redirection Using File Descriptors . . . . . | 63   |
| G. Conditional Substitution . . . . .                        | 64   |
| H. Invocation Flags . . . . .                                | 65   |
| EXAMPLES OF SHELL PROCEDURES . . . . .                       | 65   |
| EFFECTIVE AND EFFICIENT SHELL PROGRAMMING . . . . .          | 72   |
| A. Overall Approach . . . . .                                | 72   |
| B. Approximate Measures of Resource Consumption . . . . .    | 72   |
| C. Efficient Organization . . . . .                          | 73   |
| REFERENCES . . . . .                                         | 74   |
| 3. THE C PROGRAMMING LANGUAGE . . . . .                      | 75   |
| INTRODUCTION . . . . .                                       | 75   |
| C LANGUAGE . . . . .                                         | 76   |



| CONTENTS                              | PAGE |
|---------------------------------------|------|
| LEXICAL CONVENTIONS . . . . .         | 76   |
| A. Comments . . . . .                 | 76   |
| B. Identifiers (Names) . . . . .      | 76   |
| C. Keywords . . . . .                 | 76   |
| D. Constants . . . . .                | 76   |
| E. Strings . . . . .                  | 77   |
| F. Hardware Characteristics . . . . . | 77   |
| SYNTAX NOTATION . . . . .             | 78   |
| NAMES . . . . .                       | 78   |
| OBJECTS AND LVALUES . . . . .         | 79   |
| CONVERSIONS . . . . .                 | 79   |
| A. Characters and Integers . . . . .  | 80   |
| B. Float and Double . . . . .         | 80   |
| C. Floating and Integral . . . . .    | 80   |
| D. Pointers and Integers . . . . .    | 80   |
| E. Unsigned . . . . .                 | 80   |
| F. Arithmetic Conversions . . . . .   | 80   |
| G. Void . . . . .                     | 81   |
| EXPRESSIONS . . . . .                 | 81   |
| A. Primary Expressions . . . . .      | 81   |
| B. Unary Operators . . . . .          | 82   |
| C. Multiplicative Operators . . . . . | 84   |
| D. Additive Operators . . . . .       | 84   |
| E. Shift Operators . . . . .          | 85   |
| F. Relational Operators . . . . .     | 85   |
| G. Equality Operators . . . . .       | 85   |



| CONTENTS                                                    | PAGE |
|-------------------------------------------------------------|------|
| H. Bitwise AND Operator . . . . .                           | 85   |
| I. Bitwise Exclusive OR Operator . . . . .                  | 86   |
| J. Bitwise Inclusive OR Operator . . . . .                  | 86   |
| K. Logical AND Operator . . . . .                           | 86   |
| L. Logical OR Operator . . . . .                            | 86   |
| M. Conditional Operator . . . . .                           | 86   |
| N. Assignment Operators . . . . .                           | 87   |
| O. Comma Operator . . . . .                                 | 87   |
| DECLARATIONS . . . . .                                      | 87   |
| A. Storage Class Specifiers . . . . .                       | 88   |
| B. Type Specifiers . . . . .                                | 88   |
| C. Declarators . . . . .                                    | 89   |
| D. Meaning of Declarators . . . . .                         | 89   |
| E. Structure, Union, and Enumeration Declarations . . . . . | 91   |
| F. Initialization . . . . .                                 | 93   |
| G. Type Names . . . . .                                     | 95   |
| H. Typedef . . . . .                                        | 95   |
| STATEMENTS . . . . .                                        | 96   |
| A. Expression Statement . . . . .                           | 96   |
| B. Compound Statement or Block . . . . .                    | 96   |
| C. Conditional Statement . . . . .                          | 96   |
| D. While Statement . . . . .                                | 97   |
| E. Do Statement . . . . .                                   | 97   |
| F. For Statement . . . . .                                  | 97   |
| G. Switch Statement . . . . .                               | 97   |
| H. Break Statement . . . . .                                | 98   |



| CONTENTS                                        | PAGE |
|-------------------------------------------------|------|
| I. Continue Statement . . . . .                 | 98   |
| J. Return Statement . . . . .                   | 98   |
| K. Goto Statement . . . . .                     | 99   |
| L. Labeled Statement . . . . .                  | 99   |
| M. Null Statement . . . . .                     | 99   |
| EXTERNAL DEFINITIONS . . . . .                  | 99   |
| A. External Function Definitions . . . . .      | 99   |
| B. External Data Definitions . . . . .          | 100  |
| SCOPE RULES . . . . .                           | 100  |
| A. Lexical Scope . . . . .                      | 100  |
| B. Scope of Externals . . . . .                 | 101  |
| COMPILER CONTROL LINES . . . . .                | 101  |
| A. Token Replacement . . . . .                  | 101  |
| B. File Inclusion . . . . .                     | 102  |
| C. Conditional Compilation . . . . .            | 102  |
| D. Line Control . . . . .                       | 103  |
| IMPLICIT DECLARATIONS . . . . .                 | 103  |
| TYPES REVISITED . . . . .                       | 103  |
| A. Structures and Unions . . . . .              | 103  |
| B. Functions . . . . .                          | 104  |
| C. Arrays, Pointers, and Subscripting . . . . . | 104  |
| D. Explicit Pointer Conversions . . . . .       | 105  |
| CONSTANT EXPRESSIONS . . . . .                  | 106  |
| PORTABILITY CONSIDERATIONS . . . . .            | 106  |
| ANACHRONISMS . . . . .                          | 107  |
| SYNTAX SUMMARY . . . . .                        | 108  |



| CONTENTS                                                  | PAGE |
|-----------------------------------------------------------|------|
| A. Expressions . . . . .                                  | 108  |
| B. Declarations . . . . .                                 | 109  |
| C. Statements . . . . .                                   | 111  |
| D. External Definitions . . . . .                         | 112  |
| E. Preprocessor . . . . .                                 | 112  |
| LIBRARIES . . . . .                                       | 113  |
| A. General . . . . .                                      | 113  |
| B. The C Library . . . . .                                | 114  |
| C. The Object File Library . . . . .                      | 127  |
| D. The Math Library . . . . .                             | 129  |
| THE "cc" COMMAND . . . . .                                | 131  |
| A. General . . . . .                                      | 131  |
| B. Usage . . . . .                                        | 131  |
| A C PROGRAM CHECKER—"lint" . . . . .                      | 132  |
| A. General . . . . .                                      | 132  |
| B. Types of Messages . . . . .                            | 133  |
| C. Portability . . . . .                                  | 139  |
| A SYMBOLIC DEBUGGING PROGRAM—"sdb" . . . . .              | 140  |
| A. General . . . . .                                      | 140  |
| B. Usage . . . . .                                        | 140  |
| C. Source File Display and Manipulation . . . . .         | 143  |
| D. A Controlled Environment for Program Testing . . . . . | 145  |
| E. Machine Language Debugging . . . . .                   | 147  |
| F. Other Commands . . . . .                               | 147  |
| 4. FORTRAN . . . . .                                      | 149  |
| INTRODUCTION . . . . .                                    | 149  |



| CONTENTS                                           | PAGE |
|----------------------------------------------------|------|
| FORTRAN 77 . . . . .                               | 149  |
| A. General . . . . .                               | 149  |
| B. Language Extensions . . . . .                   | 150  |
| C. Violations of the Standard . . . . .            | 153  |
| D. Interprocedure Interface . . . . .              | 154  |
| E. File Formats . . . . .                          | 156  |
| A RATIONAL FORTRAN PREPROCESSOR—"ratfor" . . . . . | 157  |
| A. General . . . . .                               | 157  |
| B. Usage . . . . .                                 | 157  |
| C. Statement Grouping . . . . .                    | 158  |
| D. The "if-else" Construction . . . . .            | 159  |
| E. The "switch" Statement . . . . .                | 160  |
| F. The "do" Statement . . . . .                    | 160  |
| G. The "break" and "next" Statements . . . . .     | 161  |
| H. The "while" Statement . . . . .                 | 161  |
| I. The "for" Statement . . . . .                   | 162  |
| J. The "repeat-until" Statement . . . . .          | 163  |
| K. The "return" Statement . . . . .                | 163  |
| L. The "define" Statement . . . . .                | 163  |
| M. The "include" Statement . . . . .               | 164  |
| N. Free-Form Input . . . . .                       | 164  |
| O. Translations . . . . .                          | 164  |
| P. Warnings . . . . .                              | 165  |



## 1. INTRODUCTION

This volume describes the three main programming languages supported on the UNIX operating system. These languages are:

- **Shell**—The **shell** language is both a command language (which handles commands entered from a terminal) and a programming language (where commands are specified in a file and the file is executed).
- **C Language**—A medium-level programming language which was used to write most of the UNIX operating system. This volume describes the grammar and usage of C language, the libraries that provide additional routines, the **cc(1)** command, and two programs that are useful for checking/debugging C programs.
- **Fortran**—Fortran 77 and a rational Fortran preprocessor (**ratfor**) are available. This volume describes how Fortran 77 is implemented in terms of the variations from the American National Standard and the interfaces to the UNIX operating system. The **ratfor** preprocessor provides a means by which Fortran 77 can be written in a fashion similar to C language. This preprocessor provides (among other things) simplified control-flow statements.

Throughout this volume, each reference of the form **name(1M)**, **name(7)**, or **name(8)** refers to entries in the UNIX System Administrator's Manual. All other references to entries of the form **name(N)**, where "N" is a number (1 through 6) possibly followed by a letter, refer to entry **name** in section N of the UNIX System User's Manual.



**NOTES**



## 2. AN INTRODUCTION TO SHELL

### INTRODUCTION

The **shell** is a command programming language that provides an interface to the UNIX operating system. Its features include control-flow primitives, parameter passing, variables, and string substitution. Constructs such as **while**, **if then else**, **case**, and **for** are available. Two-way communication is possible between the **shell** and commands. String-valued parameters, typically file names or flags, may be passed to a command. A return code is set by commands that may be used to determine control-flow, and the standard output from a command may be used as **shell** input.

The **shell** can modify the environment in which commands run. Input and output can be redirected to files, and processes that communicate through **pipes** can be invoked. Commands are found by searching directories in the file system in a sequence that can be defined by the user. Commands can be read either from the terminal or from a file which allows command procedures to be stored for later use.

The **shell** is both a command language and a programming language that provides an interface to the UNIX operating system. This volume describes, with examples, the UNIX operating system **shell**. The "SIMPLE COMMANDS" part of this section covers most of the everyday requirements of terminal users. Some familiarity with the UNIX operating system is an advantage when reading this section; refer to section "BASICS FOR BEGINNERS" in the UNIX System User's Guide. The "SHELL PROCEDURES" part of this section describes those features of the **shell** primarily intended for use within **shell** commands or procedures. These include the control-flow primitives and string-valued variables provided by the **shell**. A knowledge of a programming language would also be helpful when reading this section. The last part, "KEYWORD PARAMETERS", describes the more advanced features of the **shell**. See Table 2.A for a defined listing of grammar words used in this section.

Throughout this section, each reference of the form **name**(1M), **name**(7), or **name**(8) refers to entries in the UNIX System Administrator's Manual. All other references to entries of the form **name**(N), where "N" is a number (1 through 6) possibly followed by a letter, refer to entry **name** in section N of the UNIX System User's Manual.

### SIMPLE COMMANDS

Simple commands consist of one or more words separated by blanks. The first word is the **name** of the command to be executed; any remaining words are passed as *arguments* to the command. For example,

```
who
```

is a command that prints the names of users logged in. The command

```
ls -l
```

prints a list of files in the current directory. The argument **-l** tells **ls**(1) to print status information, size, and the creation date for each file.

#### A. Background Commands

To execute a command, the **shell** normally creates a new process and waits for it to finish. A command may be run without waiting for it to finish. For example,

```
cc pgm.c &
```

calls the C compiler to compile the file *pgm.c*. The trailing "&" is an operator that instructs the **shell** not to wait for the command to finish. To help keep track of such a process, the **shell** reports its process number following its creation. A list of currently active processes may be obtained using the **ps**(1) command.



## B. Input/Output Redirection

Most commands produce output to the standard output that is initially connected to the terminal. This output may be directed to a file by using the notation ">", for example:

```
ls -l >file
```

The notation >*file* is interpreted by the **shell** and is not passed as an argument to **ls**(1). If *file* does not exist, the **shell** creates it; otherwise, the original contents of *file* are replaced with the output from **ls**(1). Output may be appended to a file using the notation ">>" as follows:

```
ls -l >>file
```

In this case, *file* is also created if it does not already exist.

The standard input of a command may be taken from a file instead of the terminal by using the notation "<", for example:

```
wc <file
```

The command **wc**(1) reads its standard input (in this case redirected from *file*) and prints the number of characters, words, and lines found. If only the number of lines is required, then

```
wc -l <file
```

can be used.

## C. Pipelines and Filters

The standard output of one command may be connected to the standard input of another by writing the "pipe" operator, indicated by |, between commands as in

```
ls -l | wc
```

Two or more commands connected in this way constitute a **pipeline**, and the overall effect is the same as

```
ls -l >file; wc <file
```

except that no *file* is used. Instead the two processes are connected by a pipe [see **pipe**(2)] and are run in parallel. Pipes are unidirectional, and synchronization is achieved by halting **wc**(1) when there is nothing to read and halting **ls**(1) when the pipe is full.

A **filter** is a command that reads its standard input, transforms it in some way, and prints the result as output. One such filter, **grep**(1) selects from its input those lines that contain some specified string. For example,

```
ls | grep old
```

prints those lines, if any, of the output from **ls** that contain the string "old". Another useful filter is **sort**(1). For example,

```
who | sort
```

will print an alphabetically sorted list of logged-in users.



A pipeline may consist of more than two commands, for example,

```
ls | grep old | wc -l
```

prints only the number of file names in the current directory containing the string "old".

#### D. File Name Generation

Many commands accept arguments which are file names. For example,

```
ls -l main.c
```

prints only information relating to the file *main.c*. The "ls -l" command alone prints the same information about all files in the current directory.

The **shell** provides a mechanism for generating a list of file names that match a pattern. For example,

```
ls -l *.c
```

generates as arguments to `ls(1)` all file names in the current directory that end in *.c*. The character "\*" is a pattern that will match any string including the null string. In general, patterns are specified as follows:

- \* Matches any string of characters including the null string.
- ? Matches any single character.
- [...] Matches any one of the characters enclosed. A pair of characters separated by a minus will match any character lexically between the pair.

For example,

```
[a-z]*
```

matches all names in the current directory beginning with one of the letters *a* through *z*. The input

```
/usr/fred/test/?
```

matches all names in the directory */usr/fred/test* that consist of a single character. If no file name is found that matches the pattern then the pattern is passed, unchanged, as an argument.

This mechanism is useful both to save typing and to select names according to some pattern. It may also be used to find files. For example,

```
echo /usr/fred/*/core
```

finds and prints the names of all *core* files in subdirectories of */usr/fred*. [The `echo(1)` command is a standard UNIX operating system command that prints its arguments, separated by blanks.] This last feature can be expensive requiring a scan of all subdirectories of */usr/fred*.

There is one exception to the general rules given for patterns. The character "." at the start of a file name must be explicitly matched. The input

```
echo *
```

will therefore echo all file names in the current directory not beginning with ".". The input

```
echo .*
```



will echo all those file names that begin with ".". This avoids inadvertent matching of the names "." and ".." which mean "the current directory" and "the parent directory", respectively. [Notice that `ls(1)` suppresses information for the files "." and "..".]

#### E. Quoting

Characters that have a special meaning to the **shell**, such as

```
< > * ? | & .
```

are called metacharacters. A complete list of metacharacters is given in Table 2.B. Any character preceded by a `\` is quoted and loses its special meaning, if any. The `\` is elided so that

```
echo \?
```

will echo a single `?`, and

```
echo \\\
```

will echo a single `\`. To allow long strings to be continued over more than one line, the sequence `\new-line` (or `RETURN`) is ignored. The `\` is convenient for quoting single characters. When more than one character needs quoting, the above mechanism is clumsy and error prone. A string of characters may be quoted by enclosing the string between single quotes. For example,

```
echo xx'****'xx
```

will echo

```
xx****xx
```

The quoted string may not contain a single quote but may contain new-lines which are preserved. This quoting mechanism is the most simple and is recommended for casual use. A third quoting mechanism using double quotes is also available and prevents interpretation of some but not all metacharacters. Details of quoting are described under "D. Evaluation and Quoting" in part "KEYWORD PARAMETERS".

#### F. Prompting by the Shell

When the **shell** is used from a terminal, it will issue a prompt to the terminal user indicating it is ready to read a command from the terminal. By default, this prompt is `"$"`. The prompt may be changed by entering,

```
PS1=newprompt
```

which sets the prompt to be the string " newprompt". If a new-line is typed and further input is needed, the **shell** will issue the prompt `">"`. Sometimes this can be caused by mistyping a quote mark. If it is unexpected, then an interrupt (`DEL`) will return the **shell** to read another command. The other prompt (`>`) may be changed (for example) by entering:

```
PS2=more
```

#### G. The Shell and Login

Following the user's `login(1)`, the **shell** is called to read and execute commands typed at the terminal. If the user's login directory contains the file `.profile`, then it is assumed to contain commands and is read immediately by the **shell** before reading any commands from the terminal.



## H. Summary

**ls**

Prints the names of files in the current directory.

**ls >file**

Puts the output from **ls** into *file*.

**ls | wc -l**

Prints the number of files in the current directory.

**ls | grep old**

Prints those file names containing the string "old".

**ls | grep old | wc -l**

Prints the number of files whose name contains the string "old".

**cc pgm.c &**

Runs **cc** in the background.

## SHELL PROCEDURES

The **shell** may be used to read and execute commands contained in a file. For example, the following call

**sh file [ args ... ]**

calls the **shell** to read commands from *file*. Such a file call is called a "command procedure" or "shell procedure". Arguments may be supplied with the call and are referred to in *file* using the positional parameters **\$1**, **\$2**, ... . For example, if the file *wg* contains

**who | grep \$1**

then the call

**sh wg fred**

is equivalent to

**who | grep fred**

All UNIX operating system files have three independent attributes (often called "permissions"), **read**, **write**, and **execute** (rwx). The UNIX operating system command **chmod(1)** may be used to make a file executable. For example,

**chmod +x wg**

will ensure that the file *wg* has execute status (permission). Following this, the command

**wg fred**

is equivalent to the call

**sh wg fred**

This allows **shell** procedures and programs to be used interchangeably. In either case, a new process is created to execute the command.



As well as providing names for the positional parameters, the number of positional parameters in the call is available as \$#. The name of the file being executed is available as \$0.

A special shell parameter \$\* is used to substitute for all positional parameters except \$0. A typical use of this is to provide some default arguments, as in,

```
nroff -T450 -cm $*
```

which simply prepends some arguments to those already given.

#### A. Control Flow—"for"

A frequent use of shell procedures is to loop through the arguments (\$1, \$2, ...) executing commands once for each argument. An example of such a procedure is *tel* that searches the file */usr/lib/telnet* that contains lines of the form

```
...
fred mh0123
bert mh0789
...
```

The text of *tel* is

```
for i
do
    grep $i /usr/lib/telnet
done
```

The command

```
tel fred
```

prints those lines in */usr/lib/telnet* that contain the string "fred".

The command

```
tel fred bert
```

prints those lines containing "fred" followed by those for "bert".

The **for** loop notation is recognized by the **shell** and has the general form

```
for name in w1 w2
do
    command-list
done
```

A **command-list** is a sequence of one or more simple commands separated or terminated by a new-line or a semicolon. Furthermore, reserved words like **do** and **done** are only recognized following a new-line or semicolon. A **name** is a **shell** variable that is set to the words *w1 w2 ...* in turn each time the **command-list** following **do** is executed. If "in *w1 w2 ...*" is omitted, then the loop is executed once for each positional parameter; that is, in \$\* is assumed.

Another example of the use of the **for** loop is the **create** command whose text is

```
for i do >$i; done
```



The command

```
create alpha beta
```

ensures that two empty files *alpha* and *beta* exist and are empty. The notation *>file* may be used on its own to create or clear the contents of a file. Notice also that a semicolon (or new-line) is required before *done*.

#### B. Control Flow — "case"

A multiple way (choice) branch is provided for by the **case** notation. For example,

```
case $# in
  1) cat >>$1 ;;
  2) cat >>$2 <$1 ;;
  *) echo 'usage: append [ from ] to' ;;
esac
```

is an append command. (Note the use of semicolons to delimit the cases.) When called with one argument as in

```
append file
```

*\$#* is the string "1". The standard input is appended (copied) onto the end of *file* using the *cat*(1) command, and

```
append file1 file2
```

appends the contents of *file1* onto *file2*. If the number of arguments supplied to *append* is other than 1 or 2, then a message is printed indicating proper usage.

The general form of the **case** command is

```
case word in
  pattern) command-list ;;
  ...
esac
```

The shell attempts to match word with each pattern in the order in which the patterns appear. If a match is found, the associated **command-list** is executed and execution of the **case** is complete. Since **\*** is the pattern that matches any string, it can be used for the default case.

**Caution:** *No check is made to ensure that only one pattern matches the case argument.*

The first match found defines the set of commands to be executed. In the example below, the commands following the second **"\*"** will never be executed since the first **"\*"** executes everything it receives.

```
case $# in
  *) ... ;;
  *) ... ;;
esac
```



Another example of the use of the **case** construction is to distinguish between different forms of an argument. The following example is a fragment of a **cc(1)** command.

```
for i
do
    case $i in
        -[ocs]    ... ;;
        -*)      echo 'unknown flag $i' ;;
        *.c)     /lib/c0 $i ... ;;
        *)      echo 'unexpected argument $i' ;;
    esac
done
```

To allow the same commands to be associated with more than one pattern, the **case** command provides for alternative patterns separated by a **|**. For example,

```
case $i in
    -x | -y)    ...
esac
```

is equivalent to

```
case $i in
    -[xy])      ...
esac
```

The usual quoting conventions apply so that

```
case $i in
    \?)        ...
```

will match the character **?**.

### C. Here Documents

The **shell** procedure **tel** described in subpart "A. Control Flow—**for**" uses the file **/usr/lib/telno** to supply the data for **grep(1)**. An alternative is to include this data within the **shell** procedure as a *here* document, as in,

```
for i
do
    grep $i <<!
    ...
    fred mh0123
    bert mh0789
    ...
!
done
```

In this example, the **shell** takes the lines between **<<!** and **!** as the standard input for **grep(1)**. The string **"!"** is arbitrary. The document is being terminated by a line that consists of the string following **<<**.



Parameters are substituted in the document before it is made available to **grep(1)** as illustrated by the following procedure called *edg*.

```
ed $3 <<%
g/$1/s//$2/g
w
%
```

The call

```
edg string1 string2 file
```

is then equivalent to the command

```
ed file <<%
g/string1/s//string2/g
w
%
```

and changes all occurrences of "string1" in *file* to "string2". Substitution can be prevented using \ to quote the special character \$ as in

```
ed $3 <<+
1,\ $s/$1/$2/g
w
+
```

[This version of *edg* is equivalent to the first except that **ed(1)** will print a ? if there are no occurrences of the string \$1.] Substitution within a *here* document may be prevented entirely by quoting the terminating string, for example,

```
grep $i <<\#
...
#
```

The document is presented without modification to **grep**. If parameter substitution is not required in a *here* document, this latter form is more efficient.

#### D. Shell Variables

The **shell** provides string-valued variables. Variable names begin with a letter and consist of letters, digits, and underscores. Variables may be given values by writing

```
user=fred box=m000 acct=mh0000
```

which assigns values to the variables *user*, *box*, and *acct*. A variable may be set to the null string by entering

```
null=
```

The value of a variable is substituted by preceding its name with \$; for example,

```
echo $user
```

will echo *fred*.

Variables may be used interactively to provide abbreviations for frequently used strings. For example,

```
b=/usr/fred/bin
mv file $b
```



will move the *file* from the current directory to the directory */usr/fred/bin*. A more general notation is available for parameter (or variable) substitution, as in,

```
echo ${user}
```

which is equivalent to

```
echo $user
```

and is used when the parameter name is followed by a letter or digit. For example,

```
tmp=/tmp/ps  
ps a >${tmp}a
```

will direct the output of *ps*(1) to the file */tmp/psa*, whereas,

```
ps a >$tmpa
```

would cause the value of the variable *tmpa* to be substituted.

Except for *\$?*, the following are set initially by the **shell**. The *\$?* is set after executing each command.

- \$?** The exit status (return code) of the last command executed as a decimal string. Most commands return a zero exit status if they complete successfully; otherwise, a nonzero exit status is returned. Testing the value of return codes is dealt with later under **if** and **while** commands.
- \$#** The number of positional parameters (in decimal). The *\$#* is used, for example, in the **append** command to check the number of parameters.
- \$\$** The process number of this **shell** (in decimal). Since process numbers are unique among all existing processes, this string is frequently used to generate unique temporary file names. For example,
- ```
ps a >/tmp/ps$$  
...  
rm /tmp/ps$$
```
- \$!** The process number of the last process run in the background (in decimal).
- \$-** The current **shell** flags, such as **-x** and **-v**.

Some variables have a special meaning to the **shell** and should be avoided for general use.

- \$MAIL** When used interactively, the **shell** looks at the file specified by this variable before it issues a prompt. If the specified file has been modified since it was last looked at, the **shell** prints the message "you have mail" before prompting for the next command. This variable is typically set in the file *.profile* in the user's login directory. For example:

```
MAIL=/usr/mail/fred
```

- \$HOME** The default argument for the *cd*(1) command. The current directory is used to resolve file name references that do not begin with a */* and is changed using the **cd** command. For example,

```
cd /usr/fred/bin
```



makes the current directory `/usr/fred/bin`. Then

```
cat wn
```

will print on the terminal the file `wn` in this directory. The command `cd(1)` with no argument is equivalent to

```
cd $HOME
```

This variable is also typically set in the user's login profile.

#### `$PATH`

A list of directories containing commands (the *search path*). Each time a command is executed by the shell, a list of directories is searched for an executable file. If `$PATH` is not set, the current directory, `/bin`, and `/usr/bin` are searched by default. Otherwise, `$PATH` consists of directory names separated by a colon (:). For example,

```
PATH=:/usr/fred/bin:/bin:/usr/bin
```

specifies that the current directory (the null string before the first :), `/usr/fred/bin`, `/bin`, and `/usr/bin` are to be searched in that order. In this way, individual users can have their own "private" commands that are accessible independently of the current directory. If the command name contains a /, this directory search is not used; a single attempt is made to execute the command.

#### `$PS1`

The primary shell prompt string, by default, `"$ "`.

#### `$PS2`

The shell prompt when further input is needed, by default, `"> "`.

#### `$IFS`

The set of characters used by *blank interpretation* (See "D. Evaluation and Quoting" in part "KEYWORD PARAMETERS").

### E. The "test" Command

The `test` command is intended for use by shell programs. For example,

```
• test -f file
```

returns zero exit status if *file* exists and nonzero exit status otherwise. In general, `test` evaluates a predicate and returns the result as its exit status. Some of the more frequently used `test` arguments are given below [see `test(1)` for a complete specification].

|                           |                                                         |
|---------------------------|---------------------------------------------------------|
| <code>test s</code>       | true if the argument <i>s</i> is<br>not the null string |
| <code>test -f file</code> | true if <i>file</i> exists                              |
| <code>test -r file</code> | true if <i>file</i> is readable                         |
| <code>test -w file</code> | true if <i>file</i> is writable                         |
| <code>test -d file</code> | true if <i>file</i> is a directory                      |

### F. Control Flow—"while"

The actions of the `for` loop and the `case` branch are determined by data available to the shell. A `while` or `until` loop and an `if then else` branch are also provided whose actions are determined by the exit status returned by commands. A `while` loop has the general form

```
while command-list1
do
    command-list2
done
```



The value tested by the **while** command is the exit status of the last simple command following **while**. Each time around the loop, *command-list1* is executed; if a zero exit status is returned, then *command-list2* is executed; otherwise, the loop terminates. For example,

```
while test $1
do
    ...
    shift
done
```

is equivalent to

```
for i
do
    ...
done
```

The **shift** command is a **shell** command that renames the positional parameters **\$2**, **\$3**, ... as **\$1**, **\$2**, ... and loses **\$1**.

Another kind of use for the **while/until** loop is to wait until some external event occurs and then run some commands. In an **until** loop, the termination condition is reversed. For example,

```
until test -f file
do
    sleep 300
done
commands
```

will loop until *file* exists. Each time around the loop, it waits for 5 minutes (300 seconds) before trying again. (Presumably, another process will eventually create the file.)

#### G. Control Flow—"if"

Also available is a general conditional branch of the form,

```
if command-list
then
    command-list
else
    command-list
fi
```

that tests the value returned by the last simple command following **if**.

The **if** command may be used in conjunction with the **test** command to test for the existence of a file as in

```
if test -f file
then
    process file
else
    do something else
fi
```



An example of the use of **if**, **case**, and **for** constructions is given in "I. The Man Command" in part "SHELL PROCEDURES".

A multiple test **if** command of the form

```

if ...
then
    ...
else
    if ...
    then
        ...
    else
        if ...
        ...
        fi
    fi
fi

```

may be written using an extension of the **if** notation as,

```

if ...
then
    ...
elif ...
then
    ...
elif ...
...
fi

```

The **touch** command changes the "last modified" time for a list of files. The command may be used in conjunction with **make**(1) to force recompilation of a list of files. The following example is the **touch** command:

```

flag=
for i
do
    case $i in
        -c)    flag=N ;;
        *)    if test -f $i
                then
                    ln $i junk$$
                    rm junk$$
                elif test $flag
                then
                    echo file \"$i\" does not exist
                else
                    >$i
                fi ;;
    esac
done

```



The `-c` flag is used in this command to force subsequent files to be created if they do not already exist. Otherwise, if the file does not exist, an error message is printed. The shell variable *flag* is set to some non-null string if the `-c` argument is encountered. The commands

```
ln ...; rm ...
```

make a link to the file and then remove it.

The sequence

```
if command1
then    command2
fi
```

may be written

```
command1 && command2
```

Conversely,

```
command1 || command2
```

executes **command2** only if **command1** fails. In each case, the value returned is that of the last simple command executed.

### Command Grouping

Commands may be grouped in two ways,

```
{ command-list ; }
```

and

```
( command-list )
```

The first form, *command-list*, is simply executed. The second form executes *command-list* as a separate process. For example,

```
( cd x; rm junk )
```

executes **rm junk** in the directory *x* without changing the current directory of the invoking shell.

The commands

```
cd x; rm junk
```

have the same effect but leave the invoking shell in the directory *x*.

### H. Debugging Shell Procedures

The shell provides two tracing mechanisms to help when debugging shell procedures. The first is invoked within the procedure as

```
set -v
```



(*v* for verbose) and causes lines of the procedure to be printed as they are read. It is useful to help isolate syntax errors. It may be invoked without modifying the procedure by entering

```
sh -v proc ...
```

where *proc* is the name of the shell procedure. This flag may be used in conjunction with the *-n* flag which prevents execution of subsequent commands. (Note that typing "*set -n*" at a terminal will render the terminal useless until an end-of-file is typed.)

The command

```
set -x
```

will produce an execution trace with flag *-x*. Following parameter substitution, each command is printed as it is executed. (Try the above at the terminal to see what effect they have.) Both flags may be turned off by typing

```
set -
```

and the current setting of the shell flags is available as *\$-*.

#### 1. The "man" Command

The following is the *man* command which is used to print sections of the UNIX System User's Manual. It is called by entering

```
man sh
man -t ed
man 2 fork
```

In the first call, the manual section for *sh* is printed. Since no section is specified, Section 1 is used. The second call will typeset (*-t* option) the manual section for *ed*. The last call prints the *fork* manual page from Section 2 of the manual.

A version of the *man* command follows:

```
cd /usr/man
: 'colon is the comment command'
: 'default is nroff ($N), section 1 ($s)'
N=n s=1
for i
do
    case $i in
        [1-9]*) s=$i ;;
        -t) N=t ;;
        -n) N=n ;;
        -*) echo unknown flag \'$i\' ;;
        *) if test -f man$s/$i.$s
            then
                ${N}roff man0/${N}aa man$s/$i.$s
            else
                : 'look through all manual sections'
```



```

        found=no
        for j in 1 2 3 4 5 6 7 8 9
        do
            if test -f man$j/$i.$j
            then man $j $i
                found=yes
            fi
        done
        case $found in
            no) echo '$i: manual page not found'
        esac
    fi ;;
done
esac

```

## KEYWORD PARAMETERS

Shell variables may be given values by assignment or when a **shell** procedure is invoked. An argument to a **shell** procedure of the form *name=value* that precedes the command name causes *value* to be assigned to *name* before execution of the procedure begins. The value of *name* in the invoking **shell** is not affected. For example,

```
user=fred command
```

will execute **command** with *user* set to *fred*. The **-k** flag causes arguments of the form *name=value* to be interpreted in this way anywhere in the argument list. Such *names* are sometimes called keyword parameters. If any arguments remain, they are available as positional parameters **\$1**, **\$2**, ... .

The **set** command may also be used to set positional parameters from within a procedure. For example,

```
set - *
```

will set **\$1** to the first file name in the current directory, **\$2** to the next, etc. Note that the first argument, **-**, ensures correct treatment when the first file name begins with a **-**.

### A. Parameter Transmission

When a **shell** procedure is invoked, both positional and keyword parameters may be supplied with the call. Keyword parameters are also made available implicitly to a **shell** procedure by specifying in advance that such parameters are to be exported. For example,

```
export user box
```

marks the variables *user* and *box* for export. When a **shell** procedure is invoked, copies are made of all exportable variables for use within the invoked procedure. Modification of such variables within the procedure does not affect the values in the invoking **shell**. It is generally true of a **shell** procedure that it may not modify the state of its caller without explicit request on the part of the caller. (Shared file descriptors are an exception to this rule.)

Names whose value is intended to remain constant may be declared readonly. The form of this command is the same as that of the **export** command,

```
readonly name ...
```

Subsequent attempts to set readonly variables are illegal.



**B. Parameter Substitution**

If a **shell** parameter is not set, then the null string is substituted for it. For example, if the variable *d* is not set,

```
echo $d
```

or

```
echo ${d}
```

will echo nothing. A default string may be given as in

```
echo ${d-.
```

which will echo the value of the variable *d* if it is set and "." otherwise. The default string is evaluated using the usual quoting conventions so that

```
echo ${d- '*'}
```

will echo \* if the variable *d* is not set. Similarly,

```
echo ${d-$1}
```

will echo the value of *d* if it is set and the value (if any) of *\$1* otherwise. A variable may be assigned a default value using the notation

```
echo ${d=.
```

which substitutes the same string as

```
echo ${d-.
```

and if *d* were not previously set, it will be set to the string ".". (The notation `${...=...}` is not available for positional parameters.)

If there is no sensible default, the notation

```
echo ${d?message}
```

will echo the value of the variable *d* if it has one; otherwise, *message* is printed by the **shell** and execution of the **shell** procedure is abandoned. If *message* is absent, a standard message is printed. A **shell** procedure that requires some parameters to be set might start as follows:

```
: ${user?} ${acct?} ${bin?}
```

```
...
```

Colon (:) is a command built in to the **shell** and does nothing once its arguments have been evaluated. If any of the variables *user*, *acct*, or *bin* are not set, the **shell** will abandon execution of the procedure.

**C. Command Substitution**

The standard output from a command can be substituted in a similar way to parameters. The command `pwd(1)` prints on its standard output the name of the current directory. For example, if the current directory is */usr/fred/bin*, the command

```
d=`pwd`
```



is equivalent to

```
d=/usr/fred/bin
```

The entire string between ('...') is taken as the command to be executed and is replaced with the output from the command. The command is written using the usual quoting conventions except that a ' must be escaped using a \. For example,

```
ls `echo " $1" `
```

is equivalent to

```
ls $1
```

Command substitution occurs in all contexts where parameter substitution occurs (including *here* documents), and the treatment of the resulting text is the same in both cases. This mechanism allows string processing commands to be used within *shell* procedures. An example of such a command is **basename** which removes a specified suffix from a string. For example,

```
basename main.c .c
```

will print the string "main". Its use is illustrated by the following fragment from a **cc(1)** command.

```
case $A in
    ...
    *.c)      B=`basename $A .c`
    ...
esac
```

that sets **B** to the part of **\$A** with the suffix **.c** stripped.

Here are some composite examples.

- for *i* in `ls -t`; do ...

The variable *i* is set  
to the names of files in time order,  
most recent first.

- set `date`; echo \$6 \$2 \$3, \$4

will print, e.g.,  
1977 Nov 1, 23:59:59

#### D. Evaluation and Quoting

The **shell** is a macro processor that provides parameter substitution, command substitution, and file name generation for the arguments to commands. This section discusses the order in which these evaluations occur and the effects of the various quoting mechanisms.

Commands are parsed initially according to the grammar given in Table 2.A. Before a command is executed, the following substitutions occur:

1. parameter substitution, e.g., **\$user**



## 2. command substitution, e.g., 'pwd'

Only one evaluation occurs so that if, for example, the value of the variable *X* is the string "\$y" then

```
echo $X
```

will echo "\$y".

## 3. blank interpretation

Following the above substitutions, the resulting characters are broken into nonblank words (*blank interpretation*). For this purpose, "blanks" are the characters of the string "\$IFS". By default, this string consists of blank, tab, and new-line. The null string is not regarded as a word unless it is quoted. For example,

```
echo ''
```

will pass on the null string as the first argument to `echo`, whereas

```
echo $null
```

will call `echo` with no arguments if the variable *null* is not set or set to the null string.

## 4. file name generation

Each word is then scanned for the file pattern characters \*, ?, and [...]; and an alphabetical-list of file names is generated to replace the word. Each such file name is a separate argument.

The evaluations just described also occur in the list of words associated with a `for` loop. Only substitution occurs in the *word* used for a `case` branch.

As well as the quoting mechanisms described earlier using \ and '...', a third quoting mechanism is provided using double quotes. Within double quotes, parameter and command substitution occurs; but file name generation and the interpretation of blanks does not. The following characters have a special meaning within double quotes and may be quoted using \.

|    |                                        |
|----|----------------------------------------|
| \$ | parameter substitution                 |
| `  | command substitution                   |
| "  | ends the quoted string                 |
| \  | quotes the special characters \$ ` " \ |

For example,

```
echo "$x"
```

will pass the value of the variable *x* as a single argument to `echo`. Similarly,

```
echo "$*"
```

will pass the positional parameters as a single argument and is equivalent to

```
echo "$1 $2 ..."
```



The notation `$@` is the same as `$*` except when it is quoted. Inputting

```
echo "$@"
```

will pass the positional parameters, unevaluated, to `echo` and is equivalent to

```
echo "$1" "$2" ...
```

The following illustration gives, for each quoting mechanism, the `shell` metacharacters that are evaluated.

|   | metacharacter |    |   |   |   |   |
|---|---------------|----|---|---|---|---|
|   | \             | \$ | * | ' | " | , |
| ' | n             | n  | n | n | n | t |
| ' | y             | n  | n | t | n | n |
| " | y             | y  | n | y | t | n |

t = terminator  
y = interpreted  
n = not interpreted

In cases where more than one evaluation of a string is required, the built-in command `eval` may be used. For example, if the variable `X` has the value `"$y"` and if `y` has the value `"pqr"`, then

```
eval echo $X
```

will echo the string `"pqr"`.

In general, the `eval` command evaluates its arguments (as do all commands) and treats the result as input to the `shell`. The input is read and the resulting command(s) executed. For example,

```
wg='eval who | grep'
$wg fred
```

is equivalent to

```
who | grep fred
```

In this example, `eval` is required since there is no interpretation of metacharacters, such as `|`, following substitution.

## E. Error Handling

The treatment of errors detected by the `shell` depends on the type of error and on whether the `shell` is being used interactively. An interactive `shell` is one whose input and output are connected to a terminal [as determined by `gtty(2)`]. A `shell` invoked with the `-i` flag is also interactive.

Execution of a command (see also "G. Command Execution") may fail for any of the following reasons:

- Input/output redirection may fail, e.g., if a file does not exist or cannot be created.



- The command itself does not exist or cannot be executed.
- The command terminates abnormally, e.g., with a "bus error" or "memory fault" signal.
- The command terminates normally but returns a nonzero exit status.

In all of these cases, the **shell** will go on to execute the next command. Except for the last case, an error message will be printed by the **shell**. All remaining errors cause the **shell** to exit from a command procedure. An interactive **shell** will return to read another command from the terminal. Such errors include the following:

- Syntax errors, e.g., if ... then ... done
- A signal such as interrupt. The **shell** waits for the current command, if any, to finish execution and then either exits or returns to the terminal.
- Failure of any of the built-in commands such as `cd(1)`.

The **shell** flag `-e` causes the **shell** to terminate if any error is detected. The following is a list of the UNIX operating system signals:

|     |                                                   |
|-----|---------------------------------------------------|
| 1   | hangup                                            |
| 2   | interrupt                                         |
| 3*  | quit                                              |
| 4*  | illegal instruction                               |
| 5*  | trace trap                                        |
| 6*  | IOT instruction                                   |
| 7*  | EMT instruction                                   |
| 8*  | floating point exception                          |
| 9   | kill (cannot be caught or ignored)                |
| 10* | bus error                                         |
| 11* | segmentation violation                            |
| 12* | bad argument to system call                       |
| 13  | write on a pipe with no one to read it            |
| 14  | alarm clock                                       |
| 15  | software termination [from <code>kill(1)</code> ] |

The UNIX operating system signals marked with an asterisk "\*" as shown in the list produce a core dump if not caught. However, the **shell** itself ignores quit which is the only external signal that can cause a dump. The signals in this list of potential interest to **shell** programs are 1, 2, 3, 14, and 15.

#### F. Fault Handling

**Shell** procedures normally terminate when an interrupt is received from the terminal. The **trap** command is used if some cleaning up is required, such as removing temporary files. For example,

```
trap 'rm /tmp/ps$$; exit' 2
```

sets a trap for signal 2 (terminal interrupt); and if this signal is received, it will execute the following commands:

```
rm /tmp/ps$$; exit
```

The **exit** is another built-in command that terminates execution of a **shell** procedure. The **exit** is required; otherwise, after the trap has been taken, the **shell** will resume executing the procedure at the place where it was interrupted.



UNIX operating system signals can be handled in one of three ways.

1. They can be ignored, in which case the signal is never sent to the process.
2. They can be caught, in which case the process must decide what action to take when the signal is received.
3. They can be left to cause termination of the process without it having to take any further action.

If a signal is being ignored on entry to the **shell** procedure, for example, by invoking it in the background (see "G. Command Execution"), **trap** commands (and the signal) are ignored.

The use of **trap** is illustrated by this modified version of the **touch** command illustrated below:

```
flag=
trap 'rm -f junk$$; exit' 1 2 3 15
for i
do
    case $i in
        -c)    flag=N ;;
        *)    if test -f $i
                then
                    ln $i junk$$; rm junk$$
                elif test $flag
                then
                    echo file \"$i\" does not exist
                else
                    >$i
                fi ;;
    esac
done
```

The cleanup action is to remove the file *junk\$\$*. The **trap** command appears before the creation of the temporary file; otherwise, it would be possible for the process to die without removing the file.

Since there is no signal 0 in the UNIX operating system, it is used by the **shell** to indicate the commands to be executed on exit from the **shell** procedure.

A procedure may, itself, elect to ignore signals by specifying the null string as the argument to **trap**. The following:

```
trap '' 1 2 3 15
```

is a fragment taken from the **nohup**(1) command which causes the UNIX operating system HANGUP, INTERRUPT, QUIT, and SOFTWARE TERMINATION signals to be ignored both by the procedure and by invoked commands.

Traps may be reset by entering

```
trap 2 3
```

which resets the traps for signals 2 and 3 to their default values. A list of the current values of traps may be obtained by writing

```
trap
```



The **scan** procedure is an example of the use of **trap** where there is no exit in the trap command. The **scan** takes each directory in the current directory, prompts with its name, and then executes commands typed at the terminal until an end of file or an interrupt is received. Interrupts are ignored while executing the requested commands but cause termination when **scan** is waiting for input. The **scan** procedure follows:

```
d='pwd'
for i in *
do
    if test -d $d/$i
    then
        cd $d/$i
        while echo " $i:" && trap exit 2 && read x
        do
            trap : 2
            eval $x
        done
    fi
done
```

The **read x** is a built-in command that reads one line from the standard input and places the result in the variable **x**. It returns a nonzero exit status if either an end-of-file is read or an interrupt is received.

#### G. Command Execution

To run a command (other than a built-in), the **shell** first creates a new process using the system call **fork(2)**. The execution environment for the command includes input, output, and the states of signals and is established in the child process before the command is executed. The built-in command **exec** is used in rare cases when no **fork** is required and simply replaces the **shell** with a new command. For example, a simple version of the **nohup** command looks like

```
trap '' 1 2 3 15
exec $*
```

The **trap** turns off the signals specified so that they are ignored by subsequently created commands, and **exec** replaces the **shell** by the command specified.

Most forms of input/output redirection have already been described. In the following, *word* is only subject to parameter and command substitution. No file name generation or blank interpretation takes place so that, for example,

```
echo ... >*.c
```

will write its output into a file whose name is *\*.c*. Input/output specifications are evaluated left to right as they appear in the command. Some input/output specifications are as follows:

- > *word*                      The standard output (file descriptor 1) is sent to the file *word* which is created if it does not already exist.
- >> *word*                     The standard output is sent to file *word*. If the file exists, then output is appended (by seeking to the end); otherwise, the file is created.
- < *word*                        The standard input (file descriptor 0) is taken from the file *word*.
- << *word*                      The standard input is taken from the lines of **shell** input that follow up to but not including a line consisting only of *word*. If *word* is quoted, no interpretation of the document occurs. If *word* is not quoted, parameter and command substitution occur and \ is used to



quote the characters `\`, `$`, ```, and the first character of *word*. In the latter case, `\new-line` is ignored (e.g., quoted strings).

- `>& digit` The file descriptor *digit* is duplicated using the system call `dup(2)`, and the result is used as the standard output.
- `<& digit` The standard input is duplicated from file descriptor *digit*.
- `<&-` The standard input is closed.
- `>&-` The standard output is closed.

Any of the above may be preceded by a digit in which case the file descriptor created is that specified by the digit instead of the default 0 or 1. For example,

```
... 2>file
```

runs a command with message output (file descriptor 2) directed to *file*. Another example,

```
... 2>&1
```

runs a command with its standard output and message output merged. (Strictly speaking, file descriptor 2 is created by duplicating file descriptor 1; but the effect is usually to merge the two streams.)

The environment for a command run in the background such as

```
list *.c | lpr &
```

is modified in two ways. First, the default standard input for such a command is the empty file `/dev/null`. This prevents two processes (the `shell` and the command), which are running in parallel, from trying to read the same input. Chaos would ensue if this were not the case. For example,

```
ed file &
```

would allow both the editor and the `shell` to read from the same input at the same time.

The other modification to the environment of a background command is to turn off the QUIT and INTERRUPT signals so that they are ignored by the command. This allows these signals to be used at the terminal without causing background commands to terminate. For this reason, the UNIX operating system convention for a signal is that if it is set to 1 (ignored) then it is never changed even for a short time. Note that the `shell` command `trap` has no effect for an ignored signal.

#### H. Invoking the Shell

The following flags are interpreted by the `shell` when it is invoked. If the first character of argument zero is a minus, commands are read from the file `.profile`.

- `-c string` If the `-c` flag is present, then commands are read from *string*.
- `-s` If the `-s` flag is present or if no arguments remain, commands are read from the standard input. `Shell` output is written to file descriptor 2.
- `-i` If the `-i` flag is present or if the `shell` input and output are attached to a terminal [as told by `getty(8)`], this `shell` is interactive. In this case, TERMINATE is ignored (so that `kill 0` does not kill an interactive `shell`, and INTERRUPT is caught and ignored (so that `wait` is interruptible). In all cases, QUIT is ignored by the `shell`.



TABLE 2.A  
GRAMMAR

|                        |                                                                                                                                                                                                                                                                                                                                                                                |
|------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>item:</i>           | <i>word</i><br><i>input-output</i><br><i>name = value</i>                                                                                                                                                                                                                                                                                                                      |
| <i>simple-command:</i> | <i>item</i><br><i>simple-command item</i>                                                                                                                                                                                                                                                                                                                                      |
| <i>command:</i>        | <i>simple-command</i><br><i>( command-list )</i><br><i>{ command-list }</i><br><i>for name do command-list done</i><br><i>for name in word ... do command-list done</i><br><i>while command-list do command-list done</i><br><i>until command-list do command-list done</i><br><i>case word in case-part ... esac</i><br><i>if command-list then command-list else-part fi</i> |
| <i>pipeline:</i>       | <i>command</i><br><i>pipeline   command</i>                                                                                                                                                                                                                                                                                                                                    |
| <i>andor:</i>          | <i>pipeline</i><br><i>andor &amp;&amp; pipeline</i><br><i>andor    pipeline</i>                                                                                                                                                                                                                                                                                                |
| <i>command-list:</i>   | <i>andor</i><br><i>command-list ;</i><br><i>command-list &amp;</i><br><i>command-list ; andor</i><br><i>command-list &amp; andor</i>                                                                                                                                                                                                                                           |
| <i>input-output:</i>   | <i>&gt; file</i><br><i>&lt; file</i><br><i>&gt;&gt; word</i><br><i>&lt;&lt; word</i>                                                                                                                                                                                                                                                                                           |
| <i>file:</i>           | <i>word</i><br><i>&amp; digit</i><br><i>&amp; -</i>                                                                                                                                                                                                                                                                                                                            |
| <i>case-part:</i>      | <i>pattern ) command-list ;;</i>                                                                                                                                                                                                                                                                                                                                               |
| <i>pattern:</i>        | <i>word</i><br><i>pattern ! word</i>                                                                                                                                                                                                                                                                                                                                           |
| <i>else-part:</i>      | <i>elif command-list then command-list else-part</i><br><i>else command-list</i>                                                                                                                                                                                                                                                                                               |
| <i>empty:</i>          | <i>empty</i>                                                                                                                                                                                                                                                                                                                                                                   |
| <i>word:</i>           | a sequence of nonblank characters                                                                                                                                                                                                                                                                                                                                              |
| <i>name:</i>           | a sequence of letters, digits, or underscores starting with a letter                                                                                                                                                                                                                                                                                                           |
| <i>digit:</i>          | 0 1 2 3 4 5 6 7 8 9                                                                                                                                                                                                                                                                                                                                                            |



TABLE 2.B

## METACHARACTERS AND RESERVED WORDS

|                            |                                                              |
|----------------------------|--------------------------------------------------------------|
| (a) <i>syntactic:</i>      |                                                              |
|                            | pipe symbol                                                  |
| &&                         | 'andf' symbol                                                |
|                            | 'orf' symbol                                                 |
| ;                          | command separator                                            |
| ::                         | case delimiter                                               |
| &                          | background commands                                          |
| ( )                        | command grouping                                             |
| <                          | input redirection                                            |
| <<                         | input from a here document                                   |
| >                          | output creation                                              |
| >>                         | output append                                                |
| (b) <i>patterns:</i>       |                                                              |
| *                          | match any character(s) including none                        |
| ?                          | match any single character                                   |
| [...]                      | match any of the enclosed characters                         |
| (c) <i>substitution:</i>   |                                                              |
| \${...}                    | substitute <b>shell</b> variable                             |
| `...`                      | substitute command output                                    |
| (d) <i>quoting:</i>        |                                                              |
| \                          | quote the next character                                     |
| '...'                      | quote the enclosed characters except for '                   |
| "..."                      | quote the enclosed characters except for the \$, ', \, and " |
| (e) <i>reserved words:</i> |                                                              |
| if then else elif fi       |                                                              |
| case in esac               |                                                              |
| for while until do done    |                                                              |
| { } [ ] test               |                                                              |



## THE SHELL TUTORIAL

### INTRODUCTION

In any programming project, some effort is used to build the end product. The remainder is consumed in building the supporting tools and procedures used to manage and maintain that end product. The second effort can far exceed the first, especially in larger projects. A good command language can be an invaluable tool in such situations. If it is a flexible programming language, it can be used to solve many internal support problems without requiring compilable programs to be written, debugged, and maintained. The most important advantage of a good command language is the ability to get the job done now. For a perspective on the motivations for using a command language in this way, see [1,2,3,4]. Throughout this section references of the type [1 through 10] refer to documents listed in part "REFERENCES".

When users log into a UNIX system, they communicate with an instance of the **shell** that reads commands typed at the terminal and arranges for the execution of the commands entered. Thus, the **shell**'s most important function is to provide a good interface for human beings. In addition, a sequence of commands may be preserved for repeated use by saving it in a file, called a *shell procedure*, *command file*, or *runcom* according to local preference.

Some UNIX system users need little knowledge of the **shell** to do their work while others make heavy use of its programming features. This section may be read in several different ways, depending on the reader's interests. A brief discussion of the UNIX system environment is found in part "OVERVIEW OF THE UNIX SYSTEM ENVIRONMENT". The discussion in part "SHELL BASICS" covers aspects of the **shell** that are important for everyone, while all of part "USING THE SHELL AS A COMMAND: SHELL PROCEDURES" and most of part "MISCELLANEOUS SUPPORTING COMMANDS AND FEATURES" are mainly of interest to those who write **shell** procedures. A group of annotated **shell** procedure examples is given in part "EXAMPLES OF SHELL PROCEDURES". Finally, a brief discussion of efficiency is offered in part "EFFECTIVE AND EFFICIENT SHELL PROGRAMMING". The discussion on efficiency is found in its proper place (at the end) and is intended for those who write especially time-consuming **shell** procedures.

Complete beginners should *not* be reading this section, but should work their way through other available tutorials first. See [10] for an appropriate plan of study.

Throughout this section, each reference of the form **name**(1M), **name**(7), or **name**(8) refers to entries in the UNIX System Administrator's Manual. All other references to entries of the form **name**(N), where "N" is a number (1 through 6) possibly followed by a letter, refer to entry **name** in section N of the UNIX System User's Manual.

### OVERVIEW OF THE UNIX SYSTEM ENVIRONMENT

Full understanding of what follows depends on familiarity with the UNIX system; [9] is useful for that, and it would be helpful to read [5] and at least one of [6,7]. For completeness, a short overview of the most relevant concepts are given below.

#### A. File System

The UNIX system file system's overall structure is that of a rooted tree composed of *directories* and other files. A simple *file name* is a sequence of characters other than a slash (/). A *pathname* is a sequence of directory names followed by a simple file name, each separated from the previous one by a /. If a pathname begins with a /, the search for the file begins at the *root* of the entire tree; otherwise, it begins at the user's *current directory* (also known as the *working directory*). The first kind of name is often called a *full* (or *absolute*) *pathname* because it is invariant with regard to the user's current directory. The latter is often called a *relative pathname*, because it specifies a path relative to the current directory. The user may change the current directory at any



time by using the `cd(1)` command. In most cases, a file name and its corresponding pathname may be used interchangeably. Some sample names are:

|                             |                                                                                                                                                                                                                                                                                                 |
|-----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>/</code>              | absolute pathname of the root directory of the entire file structure.                                                                                                                                                                                                                           |
| <code>/bin</code>           | directory containing most of the frequently used public commands.                                                                                                                                                                                                                               |
| <code>/a1/tf/jtb/bin</code> | a full (or absolute) pathname typical of multiperson programming projects. This one happens to be a private directory of commands belonging to person <i>jtb</i> in project <i>tf</i> ; <i>a1</i> is the name of a <i>file system</i> .                                                         |
| <code>bin/x</code>          | a relative pathname; it names file <i>x</i> in subdirectory <i>bin</i> of the current directory. If the current directory is <code>/</code> , it names <code>/bin/x</code> . If, on the other hand, the current directory is <code>/a1/tf/jtb</code> , it names <code>/a1/tf/jtb/bin/x</code> . |
| <code>memox</code>          | name of a file in the current directory.                                                                                                                                                                                                                                                        |

The file system for the UNIX operating system provides special shorthand notations for the current directory and the *parent* directory of the current directory:

`.` is the generic name of the current directory. A `./memox` names the same file as `memox` if such a file exists in the current directory.

`..` is the generic name of the parent directory of the current directory. If the user types:

```
cd ..
```

then the parent directory of your current working directory will become your new current directory.

## B. UNIX System Processes

An *image* is a computer execution environment, including contents of memory, register values, name of the current directory, status of open files, information recorded at login time, and various other items. A *process* is the execution of an image. Most UNIX system commands execute as separate processes. One process may spawn another using the `fork(2)` system call, which duplicates the image of the original (*parent*) process. The new (*child*) process continues to execute the same program as the parent. The two images are identical, except that each program can determine whether it is executing as parent or child. Each program may continue execution of the image or may abandon it by issuing an `exec(2)` system call, thus initiating execution of another program. In any case, each process is free to proceed in parallel with the other, although the parent most commonly issues a `wait(2)` system call to suspend execution until a child terminates (`exits`).

Figure 2.1 illustrates these ideas. **Program A** is executing (as *process 1*) and wishes to run **program B**. It `forks` and spawns a child (*process 2*) that continues to run **program A**. The child abandons A by `execing` B, while the parent goes to sleep until the child `exits`.



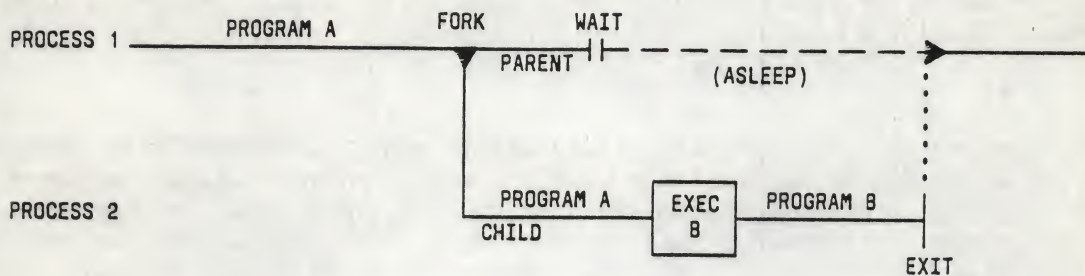


Fig. 2.1— The Shell Executing a Typical UNIX System Command

A child inherits its parent's *open files*. This mechanism permits processes to share common input streams in various ways. In particular, an open file possesses a *pointer* that indicates a position in the file and is modified by various operations on the file. The `read(2)` and `write(2)` system calls copy a requested number of bytes from and to a file beginning at the position given by the current value of the pointer. As a side effect, the pointer is incremented by the number of bytes transferred yielding the effect of sequential I/O. The `lseek(2)` system call can be used to obtain random-access I/O by setting the pointer to an absolute position within the file or to a position offset either from the end of the file or from the current pointer position.

When a process terminates, it can set an 8-bit *exit status* (see  `$?`  in "Predefined Special Variables" in subpart "D. Shell Variables") that is available to its parent. This code is *usually* used to indicate success (zero) or failure (nonzero).

*Signals* indicate the occurrence of events that may have some impact on a process. A signal may be sent to a process by another process from the terminal or by the UNIX system itself. A child process inherits its parent's signals. For most signals, a process can arrange to be terminated on receipt of a signal, to ignore it completely, or to *catch* it and take appropriate action as described in "Interrupt Handling—*trap*" in subpart "D. Control Commands". For example, an INTERRUPT signal may be sent by depressing an appropriate key (*del*, *break*, or *rubout*). The action taken depends on the requirements of the specific program being executed:

- The **shell** invokes most commands in such a way that they immediately die when an interrupt is received. For example, the `pr(1)` (print) command normally dies allowing the user to terminate unwanted output.
- The **shell** *itself* ignores interrupts when reading from the terminal because it should continue execution even when the user terminates a command like `pr`.
- The editor `ed(1)` chooses to *catch* interrupts so that it can halt its current action (especially printing) without being terminated.

## SHELL BASICS

The **shell** [i.e., the `sh(1)` command] implements the command language visible to most UNIX system users. The **shell** reads input from a terminal or a file and arranges for the execution of the requested commands. It is a program written in the C language [8]. The **shell** is *not* part of the operating system but is an ordinary user program.

### A. Commands

A *simple command* is a sequence of nonblank arguments separated by blanks or tabs. The first argument (numbered *zero*) usually specifies the name of the command to be executed. Any remaining arguments, with a few exceptions, are passed as arguments to that command. A command may be as simple as:

who



which prints the login names of users who are currently logged into the system. The following line requests the `pr(1)` command to print files `a`, `b`, and `c`:

```
pr a b c
```

If the first argument of a command names a file that is *executable* (as indicated by an appropriate set of permission bits associated with that file) and is actually a compiled program, the **shell** (as parent) spawns a new (child) process that immediately executes that program. If the file is marked as being executable but is not a compiled program, it is assumed to be a shell procedure, i.e., a file of ordinary text containing shell command lines, as well as possibly lines meant to be read by other programs. In this case, the **shell** spawns another instance of itself (a *subshell*) to read the file and execute the commands included in it. The **shell** forks to do this, but no `exec` call is made. The `man(1)` command requests that entries in the on-line UNIX System User's Manual be printed on the terminal. For example, the section that describes the `who` and `pr` commands can be printed by entering the following:

```
man who pr
```

(Incidentally, the `man(1)` command itself is actually implemented as a **shell** procedure.) From the user's viewpoint, compiled programs and **shell** procedures are invoked in exactly the same way. The **shell** determines which implementation has been used rather than requiring the user to do so. This preserves the uniformity of invocation and the ease of changing the choice of implementation for a given command. The actions of the **shell** in executing any of these commands are illustrated in Fig. 2.1.

## B. How the Shell Finds Commands

The **shell** normally searches for commands in a way that permits them to be found in three distinct locations in the file structure. The **shell** first attempts to find the command (as given on the command line) in the current directory. If this fails, the **shell** prepends the string `/bin` to the name and, finally, `/usr/bin`. The effect is to search, in order, the current directory, then the directory `/bin`, and finally, `/usr/bin`. For example the `pr(1)` and `man(1)` commands are actually the files `/bin/pr` and `/usr/bin/man`, respectively. A more complex pathname may be given either to locate a file relative to the user's current directory or to access a command via an absolute pathname. If a command name *as given* begins with a `/`, `.`, or `../` (e.g., `/bin/sort` or `../cmd`), the prepending is *not* performed. Instead, a single attempt is made to execute the command as given.

This mechanism gives the user a convenient way to execute public commands and commands in or *near* the current directory as well as the ability to execute *any* accessible command regardless of its location in the file structure. Because the current directory is usually searched first, anyone can possess a private version of a public command without affecting other users. Similarly, the creation of a new public command will not affect a user who already has a private command with the same name. The particular sequence of directories searched may be changed by resetting the `PATH` variable as described in "User-defined Variables" in subpart "D. Shell Variables".

## C. Generation of Argument Lists

Command arguments are very often file names. A list of file names can be automatically generated as arguments on a command line by specifying a pattern that the **shell** matches against the file names in a directory.

Most characters in such a pattern match themselves, but there are also special *metacharacters* that may be included in a pattern. These special characters follow:

- \* Matches any string *including* the null string
- ? Matches *any one* character



- [...] Matches any sequence of characters enclosed within the square brackets. Be warned that square brackets are also used to indicate that the enclosed argument is optional. See "A. Conditional Evaluation—test" in part "MISCELLANEOUS SUPPORTING COMMANDS AND FEATURES" for more details.
- [!...] Any sequence of characters preceded by a ! and enclosed within [...] will match any one character *other* than one of the enclosed characters. Inside square brackets, a pair of characters separated by a - includes in the set all characters lexically within the inclusive range of that pair, so that [a-de] is equivalent to [abcde].

For example, the \* matches all file names in the current directory. The \*temp\* matches all file names containing string "temp". A [a-f]\* matches all file names that begin with a through f. The [!0-9] matches all single-character names other than the digits, and \*.c matches all file names ending in .c. The /a1/tf/bin/? matches all single-character file names found in /a1/tf/bin. This pattern matching capability saves much typing and, more importantly, makes it possible to organize information in large collections of small files that are named in disciplined ways.

Pattern matching has some restrictions. If the first character of a file name is a period (.), it can be matched only by an argument that literally begins with a period. If a pattern does not match any file names, then the pattern itself is returned as the result of the match, for example:

```
echo *.c
```

will print:

```
*.c
```

if the current directory contains no files ending in .c.

Directory names should not contain the characters \*, ?, [, or ] because this may cause infinite recursion during pattern matching attempts. This may be changed in a future release.

#### D. Shell Variables

The **shell** has several mechanisms for creating variables. A variable is a name representing a string value. Certain variables are usually referred to as *parameters*. *Parameters* are the variables which are normally set only on a command line. There are also *positional parameters* (see "Positional Parameters") and *keyword parameters* (see "A. A Command's Environment" in part "USING THE SHELL AS A COMMAND: SHELL PROCEDURES"). Other variables are simply names to which the user or the **shell** itself may assign string values.

**Positional Parameters:** When a **shell** procedure is invoked, the **shell** implicitly creates *positional parameters*. The argument in position zero on the command line (the name of the **shell** procedure itself) is called \$0, the first argument is called \$1, etc. The **shift** command (see "Passing Arguments to the Shell—shift" in part "USING THE SHELL AS A COMMAND: SHELL PROCEDURES") may be used to access arguments in positions numbered higher than nine.

One can explicitly force values into these positional parameters by using the **set** command:

```
set abc def ghi
```

which assigns string1("abc") to the first positional parameter (\$1), string2 to the second (\$2), and string3 to the third (\$3). For this example, **set** also *unsets* \$4, \$5, etc. even if they were previously set. The \$0 may not



be assigned a value so that it always refers to the name of the **shell** procedure or to the name of the **shell** (in the login **shell**).

**User-defined Variables:** The **shell** also recognizes alphanumeric variables to which string values may be assigned. Positional parameters may not appear on the left-hand side of an assignment statement. Positional parameters can only be set as described in "Positional Parameters". A simple assignment is of the form:

```
name = string
```

Thereafter, `$name` will yield the value "string". A *name* is a sequence of letters, digits, and underscores that begins with a letter or an underscore. Note that no spaces surround the = in an assignment statement.

More than one assignment may appear in an assignment statement, but beware since *the shell performs the assignments from right to left*. The following command line results in the variable *a* acquiring the value "abc":

```
a=$b b=abc
```

The following are examples of simple assignments. **Double** quotes around *the right-hand side* allow blanks, tabs, semicolons, and new-lines to be included in "string", while also allowing *variable substitution* (also known as *parameter substitution*) to occur. In *parameter substitution* references to positional parameters and other variable names that are prefaced by `$` are replaced by the corresponding values, if any. **Single** quotes inhibit variable substitution. Some examples follow:

```
MAIL=/usr/mail/gas
var=" echo $1 $2 $3 $4"
stars=*****
asterisks='$stars'
```

The variable *var* has as its value the string consisting of the values of the first four positional parameters, separated by blanks. No quotes are needed around the string of asterisks being assigned to **stars** because pattern matching (expansion of `*`, `?`, `[...]`) does *not* apply in this context. Note that the value of `$asterisks` is the literal string " \$stars", *not* the string "\*\*\*\*\*", because the single quotes inhibit substitution.

In assignments, blanks are not reinterpreted after variable substitution, so that the following example results in `$first` and `$second` having the same value:

```
first='a string with embedded blanks'
second=$first
```

In accessing the value of a variable, one may enclose the variable's name (or the digit designating the positional parameter) in braces `{}` to delimit the variable name from any following string. See "Command Grouping—Parentheses and Braces" in "D. Control Commands" and "G. Conditional Substitution" in part "MISCELLANEOUS SUPPORTING COMMANDS AND FEATURES" for other meanings of braces in the **shell**. In particular, if the character immediately following the name is a letter, digit, or underscore (digit only for positional parameters), then the braces are *required*:

```
a='This is a string'
echo" ${a}ent test"
```

The following variables are used by the **shell**. Some of them are set by the **shell**, and all of them can be set and reset by the user:

**HOME** is initialized by the **login**(1) program to the name of the user's *login directory*, i.e., the directory that becomes the current directory upon completion of a login. The **cd** command



without arguments uses `$HOME` as the directory to switch to. Using this variable helps one to keep full pathnames out of `shell` procedures. This is a big help when the pathname of your login directory is changed (e.g., to balance disk loads).

## **MAIL**

is the pathname of a file where your mail is deposited. If `MAIL` is set, then the `shell` checks to see if anything has been added to the file it names and announces the arrival of new mail every time you return to command level (e.g., by leaving the editor). `MAIL` must be set by the user. (The presence of mail in the standard mail file is also announced at login, regardless of whether `MAIL` is set.)

## **PATH**

is the variable that specifies where the `shell` is to look when it is searching for commands. Its value is an ordered list of directory pathnames separated by colons. A null character anywhere in that list represents the current directory. The `shell` initializes `PATH` to the list `:/bin:/usr/bin` where, by convention, a null character appears in front of the first colon. Thus if you wish to search your current directory last, rather than first, you would type:

```
PATH=/bin:/usr/bin::
```

where the two colons together represent a colon followed by a null followed by a colon, thus naming the current directory. A user often has a personal directory of commands (say, `$HOME`) and causes it to be searched *before* the `/bin` and `usr/bin` directories by using:

```
PATH=:$HOME/bin:/bin:/usr/bin
```

The setting of `PATH` to other than the default value is normally done in a user's `.profile` file (see "The `.profile` File" in subpart "I. Changing the State of the shell and the `.profile` File").

## **CDPATH**

is the variable that specifies where the `shell` is to look when searching for the argument of the `cd` command whenever that argument is not null and does not begin with `/`, `.`, or `../` [see `cd(1)`, "A. File System" in part "OVERVIEW OF THE UNIX SYSTEM ENVIRONMENT", and "E. Special Shell Commands" in part "USING THE SHELL AS A COMMAND: SHELL PROCEDURES"]. The value of `CDPATH` is an ordered list of directory pathnames separated by colons. A null character anywhere in that list represents the current directory. By convention, if the list begins with a colon, a null character is assumed to precede that colon. Initially, `CDPATH` is *unset*, resulting in only the current directory being searched. Thus if you wish the `cd` command to first search your current directory and then your home directory, you would type:

```
CDPATH=:$HOME
```

The setting of `CDPATH` to other than the default value is normally done in a user's `.profile` file (see "The `.profile` File" in subpart "I. Changing the State of the shell and the `.profile` File"). Note that if the `cd` command changes to a directory that is *not* a descendent of the current directory, it writes the full name of the new directory on the diagnostic output (see "Standard Input and Standard Output" and "Diagnostic and Other Outputs" in subpart "F. Redirection of Input and Output").

## **PS1**

is the variable that specifies what string is to be used as the primary *prompt* string. If the `shell` is interactive, it prompts with the value of `PS1` when it expects input. The default value of `PS1` is `"$"` (a `$` followed by a blank).

## **PS2**

is the variable that specifies the secondary prompt string. If the `shell` expects more input when it encounters a new-line in its input, it will prompt with the value of `PS2`. The default value of `PS2` is `">"` (a `>` followed by a blank).



**IFS** is the variable that specifies which characters are *internal field separators*. These are the characters the **shell** uses during blank interpretation. (If you want to parse some delimiter-separated data easily, you can set *IFS* to include that delimiter.) The **shell** initially sets *IFS* to include the blank, tab, and new-line characters.

**Command Substitution:** Any command line can be placed within grave accents ('...') to capture the output of the command. This concept is known as *command substitution*. The command or commands enclosed between grave accents are first executed by the **shell** and then their output replaces the whole expression, grave accents and all. This feature is often combined with **shell** variables so that

```
today=`date`
```

assigns the string representing the current date to the variable *today* (e.g., Tue Nov 27 16:01:09 EST 1979). The command

```
users=`who | wc -l`
```

saves the number of logged-in users in the variable *users*. Any command that writes to the standard output can be enclosed in grave accents. Grave accents (see "E. Quoting Mechanisms") may be nested. The inside sets must be escaped with \. For example:

```
logmsg=`echo Your login directory is `pwd``
```

Shell variables can also be given values indirectly by using the **read**(2) command. The **read** command takes a line from the standard input (usually your terminal) and assigns consecutive words on that line to any variables named:

```
read first init last
```

will take an input line of the form:

```
G. A. Snyder
```

and have the same effect as if you had typed:

```
first=G.  init=A.  last=Snyder
```

The **read** command assigns any excess "words" to the last variable.

**Predefined Special Variables:** Several variables have special meanings. The following are set *only* by the **shell**:

**\$#** records the number of *positional* arguments passed to the **shell**, not counting the name of the **shell** procedure itself. The variable **\$#** yields the number of the highest-numbered positional parameter that is set. Thus, **sh x a b c** sets **\$#** to 3. One of its primary uses is in checking for the presence of the required number of arguments:

```
if test $# -lt 2
then
    echo 'two or more args required'; exit
fi
```

**\$?** is the exit status (also referred to as *return code*, *exit code*, or *value*) of the last command executed. Its value is a decimal string. Most UNIX system commands return 0 to indicate successful completion. The **shell** itself returns the current value of **\$?** as its exit status.

**\$\$** is the process number of the current process. Since process numbers are unique among all existing processes, this string of up to five digits is often used to generate unique names



for temporary files. The UNIX system provides no mechanism for the automatic creation and deletion of temporary files. A file exists until it is explicitly removed. Temporary files are generally undesirable. The UNIX system pipe mechanism is far superior for many applications. However, the need for uniquely-named temporary files does occasionally occur. The following example also illustrates the recommended practice of creating temporary files in a directory used only for that purpose:

```
temp=$HOME/temp/$$          # use current process number
ls > $temp                  # to form unique temp file
    commands, some of which use $temp, go here
rm $temp                    # clean up at end
```

- \$!** is the process number of the last process run in the background (using **&**—see “D. Control Commands” in part “USING THE SHELL AS A COMMAND: SHELL PROCEDURES”). Again, this is a string of up to five digits.
- \$-** is a string consisting of names of execution flags (see “Execution Flags—**set**” in subpart “F. Redirection of Input and Output” and “G. More about Execution Flags” in part “USING THE SHELL AS A COMMAND: SHELL PROCEDURES”) currently turned on in the shell. The **\$-** variable might have the value **xv** if you are tracing your output.

#### E. Quoting Mechanisms

Many characters have a special meaning to the **shell** which is sometimes necessary to conceal. Single quotes (') and double quotes (") surrounding a string or backslash (\) before a single character provide this function in somewhat different ways. (Grave accents [ ` ] are sometimes called *back quotes* but are used only for command substitution [see “Command Substitution” in subpart “D. Shell Variables”] in the **shell** and do not hide special meanings of any characters.)

Within right single quotes, all characters (except ' itself) are taken literally with any special meaning removed. Thus:

```
stuff='echo $? $*; ls * | wc'
```

results only in the string **echo \$? \$\*; ls \* | wc** being assigned to the variable *stuff* but *not* in any other commands being executed.

Within double quotes, the special meaning of certain characters does persist while all other characters are taken literally. The characters that retain their special meaning are **\$**, **'**, and **"** itself. Thus, within double quotes, variables are expanded and command substitution takes place. However, any commands in a command substitution are not affected by double quotes outside of the grave accents, so that characters such as **\*** retain their special meaning.

To hide the special meaning of **\$**, **'**, and **"** within double quotes, you can precede these characters with a backslash (\). Outside of double quotes, preceding a character with \ is equivalent to placing single quotes around that character. A \ followed by a new-line causes that new-line to be ignored, thus allowing continuation of long command lines.

#### F. Redirection of Input and Output

In general, most commands neither know nor care whether their input (output) is coming from (going to) a terminal or a file. Thus, a command can be used conveniently either at a terminal or in a pipeline (see subpart



"G. Command Lines and Pipelines"). Depending on the nature of a command's input or output, the actions taken by a few commands can be varied either for efficiency's sake or to avoid useless actions (such as attempting random access I/O on a terminal).

**Standard Input/Output:** When a command begins execution, it usually expects that three files are already open—a *standard input*, a *standard output*, and a *diagnostic (error) output*. A number called a *file descriptor* is associated with each of these files. By convention, the file descriptor 0 is associated with standard input, file descriptor 1 with standard output, and file descriptor 2 with diagnostic output. A child process normally inherits these files from its parent. All three files are initially connected to the terminal (0 to the keyboard, 1 and 2 to the printer or screen). The **shell** permits them to be redirected elsewhere before control is passed to an invoked command. An argument to the **shell** of the form `< file` or `> file` opens the specified file as the standard input or output, respectively (in the case of output, destroying the previous contents of *file*, if any). An argument of the form `>> file` directs the standard output to the end of *file*, thus providing a way to *append* data to it without destroying its existing contents. In either of the two output cases, the **shell** creates *file* if it does not already exist (thus `> output` alone on a line creates a zero-length file). The following appends to file *log* the list of users who are currently logged on:

```
who >> log
```

Such redirection arguments are only subject to variable and command substitution. Neither blank interpretation nor pattern matching of file names occurs after these substitutions. Thus:

```
echo 'this is a test' > *.ggg
```

and:

```
cat < ?
```

will produce, respectively, a 1-line file named *\*.ggg* (a rather disastrous name for a file) and an error message (unless you have a file named *?*, which is also *not* a wise choice for a file name—see end of part "C. Generation of Argument Lists").

**Diagnostic & Other Outputs:** Diagnostic output from UNIX system commands is traditionally directed to the file associated with file descriptor 2. (There is often a need for an error output file that is different from standard output so that error messages do not get lost down pipelines—see subpart "G. Command Lines and Pipelines".) One can redirect this error output to a file by immediately prepending the number of the file descriptor (i.e., 2 in this case) to either output redirection symbol (`>` or `>>`). The following line will append error messages from the *cc(1)* command to file *ERRORS*:

```
cc testfile.c 2>> ERRORS
```

Note that the file descriptor number must be prepended to the redirection symbol *without* any intervening blanks or tabs. Otherwise, the number will be passed as an argument to the command.

This method may be generalized to allow one to redirect output associated with any of the first ten file descriptors (numbered 0 through 9) so that, for instance, if *cmd* puts output on file descriptor 9, the following line will capture that output in file *savedata*:

```
cmd 9> savedata
```

A command often generates standard output and error output and might even have some other output, perhaps a data file. In this case, one can redirect independently all the different outputs. Suppose that *cmd* directs its standard output to file descriptor 1, its error output to file descriptor 2, and builds a data file on file descriptor 9. The following would direct each of these three outputs to a different file:

```
cmd > standard 2> error 9> data
```



Other forms of input/output redirection are described in "Input/Output Redirection and Control Commands" and "In-line Input Documents" in subpart "D. Control Commands" and "F. Input/Output Redirection Using File Descriptors" in part "MISCELLANEOUS SUPPORTING COMMANDS AND FEATURES".

### G. Command Lines and Pipelines

A sequence of one or more commands separated by `|` (or `^`) make up a *pipeline*. In a pipeline consisting of more than one command, each command is run as a separate process connected to its neighbor(s) by *pipes*, i.e., the *output* of each command (except the last one) becomes the *input* of the next command in line. A *filter* is a command that reads its standard input, transforms it in some way, then writes it as its standard output. A pipeline normally consists of a series of filters. Although the processes in a pipeline are permitted to execute in parallel, they are synchronized to the extent that each program needs to read the output of its predecessor. Many commands operate on individual lines of text, reading a line, processing it, writing it out, and looping back for more input. Some must read larger amounts of data before producing output. The `sort(1)` command is an example of the extreme case that requires all input to be read before any output is produced.

The following is an example of a typical pipeline: `nroff` (see `troff` in Section 1) is a text formatter whose output may contain reverse line motions; `col(1)` converts these motions to a form that can be printed on a terminal lacking reverse-motion capability; `greek(1)` is used to adapt the output to a specific terminal, here specified by `-Thp`. The flag `-cm` indicates one of the commonly used formatting options, and *text* is the name of the file to be formatted:

```
nroff -cm text | col | greek -Thp
```

### H. Examples

The following examples illustrate the variety of effects that can be obtained by combining a few commands in the ways described above. It may be helpful to try these examples at a terminal:

```
who
```

Prints (on the terminal) the list of logged-in users.

```
who >> log
```

Appends the list of logged-in users to the end of file *log*.

```
who | wc -l
```

Prints the number of logged-in users. (The argument to `wc` is *minus ell*.)

```
who | pr
```

Prints a paginated list of logged-in users.

```
who | sort
```

Prints an alphabetized list of logged-in users.

```
who | grep pw
```

Prints the list of logged-in users whose login names contain the string "pw".

```
who | grep pw | sort | pr
```

Prints an alphabetized, paginated list of logged-in users whose login names contain the string "pw".

```
{ date; who | wc -l; } >> log
```

Appends (to file *log*) the current date followed by the count of logged-in users (see "Command Grouping—Parentheses and Braces" in subpart "D. Control Commands" for the meaning of {...} in this context).



```
who | sed 's/.*//' | sort | uniq -d
```

Prints only the login names of all users who are logged in more than once.

The **who** command does not *by itself* provide options to yield all of the results which can be obtained by combining **who** with other commands. Note that **who** just serves as the data source in these examples. As an exercise, replace **who** | by **< /etc/passwd** in the above examples to see how a file can be used as a data source in the same way. Notice that redirection arguments may appear anywhere on the command line.

#### 1. Changing of the Shell and .profile State

The state of a given instance of the **shell** includes the values of positional parameters (see "Positional Parameters" in subpart "D. Shell Variables"), user-defined variables (see "User-defined Variables" in subpart "D. Shell Variables"), environmental variables (see "A. A Command's Environment"), modes of execution (see "G. More about Execution Flags"), and the current working directory.

The state of a **shell** may be altered in various ways. These include the **cd** command, several flags that can be set by the user, and a file in one's login directory called **.profile** that is treated specially by the **shell**.

**"CD":** The **cd(1)** command changes the current directory to the one specified as its argument. This can (and should) be used to change to a convenient place in the directory structure. The **cd** command is often combined with **()** to cause a subshell to change to a different directory and execute a group of commands without affecting the original **shell**. The first sequence below extracts the component files of the archive file **/a1/tf/q.a** and places them in whatever directory is the current one. The second sequence of commands places them in directory **/a1/tf**.

```
ar x /a1/tf/q.a
(cd /a1/tf; ar x q.a)
```

**The .profile File:** When you log in, the **shell** is invoked to read your commands. First, however, the **shell** checks to see if a file name **/etc/profile** exists on your UNIX system and, if it does, commands are read from it. The **/etc/profile** is used by system administrators to set up variables needed by *all* users. Type

```
cat /etc/profile
```

to see what your system administrator has already done for you. After this, the **shell** proceeds to see if you have a file named **.profile** in your login directory. If so, commands are read and executed from it. For a sample **.profile**, see **profile(5)**. Finally, the **shell** is ready to read commands from your standard input—usually the terminal.

**The Execution Flags—"set":** The **set** command provides the capability of altering several aspects of the behavior of the **shell** by setting certain *shell flags*. In particular, the **x** and **v** flags may be useful from the terminal. Flags may be set by typing, for example:

```
set -xv
```

(to turn on flags **x** and **v**). The same flags may be turned *off* by typing

```
set +xv
```

These two flags have the following meaning:

|    |                                                                                                                                                                                                             |
|----|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -v | Input lines are printed as they are read by the <b>shell</b> . This flag is particularly useful for isolating syntax errors. The commands on each input line are executed after that input line is printed. |
|----|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|



**-x** Commands and their arguments are printed as they are executed. (Shell control commands, such as **for**, **while**, etc., are not printed, however.) Note that **-x** causes a trace of *only* those commands that are actually executed, whereas **-v** prints each line of input until a syntax error is detected.

The **set** command is also used to set these and other flags within **shell** procedures (see "G. More about Execution Flags").

## USING THE SHELL AS A COMMAND: SHELL PROCEDURES

### A. A Command's Environment

All the variables (with their associated values) that are known to a command at the beginning of execution of that command constitute its *environment*. This environment includes variables that the command inherits from its parent process and variables specified as *keyword parameters* on the command line that invokes the command.

The variables that a **shell** passes to its child processes are those that have been named as arguments to the **export** (see **sh**) command. The **export** command places the named variables in the environments of both the **shell** and all its future child processes.

Keyword parameters are variable-value pairs that appear in the form of assignments, normally *before* the procedure name on a command line (see also **-k** flag in "G. More about Execution Flags"). Such variables are placed in the environment of the procedure being invoked. For example:

```
#      key_command
echo $a $b
```

is a simple procedure that **echoes** the values of two variables. If it is invoked as:

```
a=key1 b=key2 key_command
```

then the output is:

```
key1 key2
```

A procedure's keyword parameters are *not* included in the argument count **\$#** (see "Predefined Special Variables" in "D. Shell Variables").

A procedure may access the value of any variable in its environment. However, if changes are made to the value of a variable, these changes are *not* reflected in the environment. The changes are local to the procedure in question. In order for these changes to be placed in the environment that the procedure passes to *its* child processes, the variable must be named as an argument to the **export** command within that procedure (see "B. Invoking the Shell"). To obtain a list of variables that have been made **exportable** from the current **shell**, type:

```
export
```

(You will also get a list of variables that have been made **readonly**—see "E. Special Shell Commands".) To get a list of name-value pairs in the current environment, type:

```
env
```



## B. Invoking the Shell

The **shell** is an ordinary command and may be invoked in the same way as other commands:

- sh proc [arg...]** A new instance of the **shell** is explicitly invoked to read **proc**. Arguments, if any, can be manipulated as described in "C. Passing Arguments to the Shell; **shift**".
- sh -v proc [arg...]** This is equivalent to putting **set -v** at the beginning of **proc**. Similarly for the **x**, **e**, **u**, and **n** flags (see "Execution Flags—**set**" in subpart "I. Changing the State of the Shell and the *profile* File" and "G. More about Execution Flags").
- proc [arg...]** If **proc** is marked executable and is not a compiled, executable program, the effect is similar to that of **sh proc [args...]**. An advantage of this form is that **proc** may be found by the search procedure described in "B. How the Shell Finds Commands" and "User-defined Variables" in subpart "D. Shell Variables". Also, variables that have been **exported** in the **shell** will still be **exported** from **proc** when this form is used (because the **shell** only forks to read commands from **proc**). Thus any changes made within **proc** to the values of **exported** variables will be passed on to subsequent commands invoked from within **proc**.

## C. Passing Arguments to the Shell—"shift"

When a command line is scanned, any character sequence of the form **\$n** is replaced by the *n*th argument to the **shell** counting the name of the **shell** procedure itself as **\$0**. This notation permits direct reference to the procedure name and to as many as nine positional parameters (see "Positional Parameters" in subpart "D. Shell Variables"). Additional arguments can be processed using the **shift** command or by using a **for** loop (see "Looping over a List—**for**" in subpart "D. Control Commands").

The **shift** command shifts arguments to the left; i.e., the value of **\$1** is thrown away, **\$2** replaces **\$1**, **\$3** replaces **\$2**, etc.. The highest-numbered positional parameter becomes *unset*. (**\$0** is *never* shifted.) The command **shift n** is a shorthand notation for *n* consecutive **shifts**. A **shift 0** does nothing. For example, consider the **shell** procedure **ripple** below. The **echo** command writes its arguments to the standard output. The **while** command is discussed in "Conditional Looping—**while** and **until**" in subpart "D. Control Commands". The lines that begin with **#** are comments.

```
#      ripple command
while test $# != 0
do
    echo $1 $2 $3 $4 $5 $6 $7 $8 $9
    shift
done
```

If the procedure were invoked by

```
ripple a b c
```

it would print

```
a b c
b c
c
```

The notation **\$\*** causes substitution of *all* positional parameters except **\$0**. Thus, the **echo** line in the **ripple** example above could be written more compactly as:

```
echo $*
```

These two **echo** commands are *not* equivalent. The first prints at most nine positional parameters. The second prints *all* of the current positional parameters. The **\$\*** notation is more concise and less error-prone. One



obvious application is in passing an arbitrary number of arguments to a command such as the `nroff` text formatter:

```
nroff -h -rW120 -T450 -cm $*
```

It is important to understand the sequence of actions used by the **shell** in scanning command lines and substituting arguments. The **shell** first reads input up to a new-line or semicolon and then parses that much of the input. Variables are replaced by their values and then command substitution (via *grave accents*) is attempted. I/O redirection arguments are detected, acted upon, and deleted from the command line. Next the **shell** scans the resulting command line for *internal field separators*, that is, for any characters specified by *IFS* to break the command line into distinct arguments. Any *explicit* null arguments (specified by `" "` or `' '`) are retained, while *implicit* null arguments resulting from evaluation of variables that are null or not set are removed. Then file name generation occurs, with all metacharacters being expanded. The resulting command line is executed by the **shell**.

Sometimes, one builds command lines inside a **shell** procedure. In this case one might want to have the **shell** rescan the command line after all the initial substitutions and expansions are done. The special command `eval` is available for this purpose. The `eval` command takes a command line as its argument and simply rescans the line performing any variable or command substitutions that are specified. Consider the following (simplified) situation:

```
command=who
output='! wc -l'
eval $command $output
```

This segment of code results in the pipeline `who ! wc -l` being executed.

The output of `eval` cannot be redirected. The uses of `eval` can, however, be nested.

#### D. Control Commands

The **shell** provides several flow-of-control commands that are useful in creating **shell** procedures. To explain them, we first need a few definitions.

A **simple command** is defined in "A. Commands". Input/Output redirection arguments can appear in a simple command line and are passed to the **shell**, *not* to the command.

A **command** is a simple command or any of the **shell** control commands described below. A **pipeline** is a sequence of one or more commands separated by `|`. (For historical reasons, `^` is a synonym for `|` in this context.) The standard output of each command but the last in a pipeline is connected [by a `pipe(2)`] to the standard input of the next command. Each command in a pipeline is run separately. The **shell** waits for the last command to finish. The exit status of a pipeline is nonzero if the exit status of either the first or last process in the pipeline is nonzero. (This is a bit weird and may be changed in the future.)

A **command list** is a sequence of one or more pipelines separated by `;`, `&`, `&&`, or `!!`, and optionally terminated by `;` or `&`. A semicolon (`;`) causes sequential execution of the previous pipeline (i.e., the **shell** waits for the pipeline to finish before reading the next pipeline), while `&` causes asynchronous execution of the preceding pipeline. Both sequential and asynchronous execution are thus allowed. An asynchronous pipeline continues execution until it terminates voluntarily or until its processes are killed. Figure 2.2 shows the actions of the **shell** involved in executing these two command lists:

```
who >log; date
who >log& date&
```

For the first command list in Fig. 2.2, the **shell** executes `who`, waits for it to terminate, then executes `date` and waits for it to terminate. For the second command list in Fig. 2.2, the **shell** invokes both commands in order but does not wait for either one to finish.



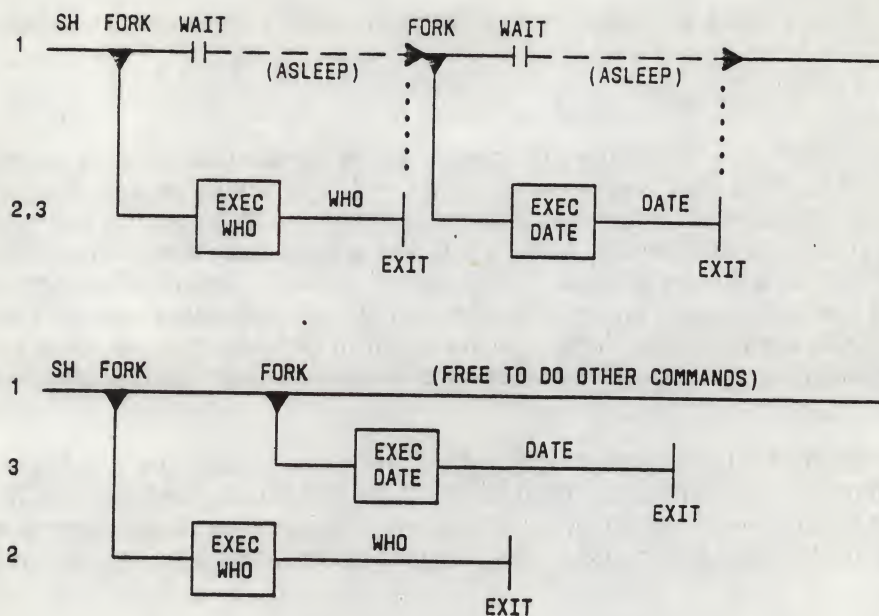


Fig. 2.2—The Shell Executing Typical Command Lists

More typical uses of & include off-line printing, background compilation, and generation of jobs to be sent to other computers. For example, if you type

```
nohup cc prog.c&
```

you may continue working while the C compiler runs in the background. A command line ending with & is immune to interrupts and quits, but it is wise to make it immune to hang-ups as well. The **nohup** command is used for this purpose. Without **nohup**, if you hang up while **cc** in the above example is still executing, **cc** will be killed and your output will disappear.

**Note:** The & operator should be used with restraint, especially on heavily-loaded systems. Other users will not consider you a good citizen if you start up a large number of simultaneous, asynchronous processes without a compelling reason for doing so.

The && and !! operators, which are of equal precedence (but lower than & and !), cause conditional execution of pipelines. In **cmd1 !! cmd2**, **cmd1** is executed and its exit status examined. Only if **cmd1** fails (i.e., has a nonzero exit status) is **cmd2** executed. This is thus a more terse notation for:

```
if cmd1
    test $? != 0
then
    cmd2
fi
```

See **writemail** in part "EXAMPLES OF SHELL PROCEDURES" for an example of use of !!.

The && operator yields the complementary test: in **cmd1 && cmd2**, the second command is executed only if the first succeeds (has a zero exit status). In the sequence below, each command is executed in order until one fails:

```
cmd1 && cmd2 && cmd3 && ... && cmdn
```



A simple command in a pipeline may be replaced by a command list enclosed in either parentheses or braces. The output of all the commands so enclosed is combined into one stream that becomes the input to the next command in the pipeline. The following line prints two separate documents in a way similar to that shown in a previous example (see "G. Command Lines and Pipelines"):

```
{ nroff -cm text1; nroff -cm text2; } | col | greek -Thp
```

See "Command Grouping—Parentheses and Braces" in subpart "D. Control Commands" for further details on command grouping.

All of the following commands are formally described in `sh(1)`.

**Structured Conditional—"if":** The shell provides an `if` command. The simplest form of the `if` command is:

```
if command list
then
    command list
fi
```

The *command list* following `if` is executed. If the last command in this list has a *zero* exit status, then the *command list* that follows `then` is executed. The `fi` indicates the end of the `if` command.

In order to cause an alternative set of commands to be executed in the case where the *command list* following `if` has a *nonzero* exit status, one may add an `else` clause to the form given above. This results in the following structure:

```
if command list
then
    command list
else
    command list
fi
```

Multiple tests can be achieved in an `if` command by using the `elif` clause. For example:

```
if test -f "$1"          # is $1 a file?
then
    pr $1
elif test -d "$1"        # else, is $1 a directory?
then
    (cd $1; pr *)
else
    echo $1 is neither a file nor a directory
fi
```

The above example is executed as follows. If the value of the first positional parameter is a file name, then print that file. If not, then check to see if it is the name of a directory. If so, change to that directory and print all the files there. Otherwise, `echo` the error message.

The `if` command may be nested (but be sure to end each one with an `fi`). The new-lines in the above examples of `if` may be replaced by semicolons.

The exit status of the `if` command is the exit status of the last command executed in any `then` clause or `else` clause. If no such command was executed, `if` returns a zero exit status.



**Multiway Branch—"case":** A multiple way branch is provided by the `case` command. The basic format of `case` is:

```
case string in
    pattern) command list ;;
.
.
.
    pattern) command list ;;
esac
```

The `shell` tries to match *string* against each pattern in turn, using the same pattern-matching conventions as in file-name generation ("C. Generation of Argument Lists"). If a match is found, the *command list* following the matched pattern is executed. The `;;` serves as a break out of the `case` and is required after each command list except the last. Note that only one pattern is ever matched and that matches are attempted in order, so that if `*` is the first pattern in a `case`, no other patterns will ever be looked at.

More than one pattern may be associated with a given command list by specifying alternate patterns separated by `|`. For example:

```
case $i in
    *.c)          cc $i
                  ;;
    *.hi*.sh)     # do nothing
                  ;;
    *)            echo "$i of unknown type"
                  ;;
esac
```

In the above example, no action is taken for the second set of patterns because the `null` command is specified. The `*` is used as a default pattern because it matches any word.

The exit status of `case` is the exit status of the last command executed in the `case` command. If no commands were executed, then `case` has a zero exit status.

**Conditional Looping—"while" and "until":** A `while` command has the general form:

```
while command list
do
    command list
done
```

The commands in the first *command list* are executed; and if the exit status of the last command in that list is zero, then the commands in the second list are executed. This sequence is repeated as long as the exit status of the first *command list* is zero. A loop can be executed as long as the first *command list* returns a nonzero exit status by replacing `while` with `until`.

Any new-line in the above example may be replaced by a semicolon. The exit status of a `while` (`until`) command is the exit status of the last command executed in the *second* command list. If no such command is executed, `while` (`until`) has exit status zero.

**Looping over a List—"for":** Often, one wishes to perform some set of operations for each in a set of files or execute some command once for each of several arguments. The `for` command can be used to accomplish this.



The **for** command has the format:

```
for variable in word list
do
    command list
done
```

where *word list* is a list of strings separated by blanks. The commands in the *command list* are executed once for each word in *word list*. *Variable* takes on as its value each word from *word list*, in turn, *word list* is fixed after it is evaluated the first time. For example, the following **for** loop will cause each of the C source files *xec.c*, *cmd.c*, and *word.c* in the current directory to be **diff**d with a file of the same name in the directory */usr/src/cmd/sh*:

```
for cfile in xec cmd word
do
    diff $cfile.c /usr/src/cmd/sh/$cfile.c
done
```

One can omit the "**in word list**" part of a **for** command. This will cause the current set of positional parameters to be used in place of *word list*. This is very convenient when one wishes to write a command that performs the same set of commands for each of an unknown number of arguments. See **null** in part "Examples of Shell Procedures" for an example of this feature.

**Loop Control—"break" and "continue"**: The **break** command can be used to terminate execution of a **while**, **until**, or a **for** loop. The **continue** command requests the execution of the next iteration of the loop. These commands are effective only when they appear between **do** and **done**.

The **break** command terminates execution of the smallest (i.e., innermost) enclosing loop causing execution to resume after the nearest following unmatched **done**. Exit from *n* levels is obtained by **break n**.

The **continue** command causes execution to resume at the nearest enclosing **while**, **until**, or **for**, i.e., the one that begins the innermost loop containing the **continue**. One can also specify an argument *n* to **continue** and execution will resume at the *n*th enclosing loop:

```
# This procedure is interactive; 'break' and 'continue'
# commands are used to allow the user to control data entry.
while true
do
    echo " Please enter data"
    read response
    case "$response" in
        " done" )      break          # no more data
                        ;;
        " " )          continue
                        ;;
        *)
                        process the data here
                        ;;
    esac
done
```

**End-of-file and "exit"**: When the **shell** reaches end-of-file, it terminates execution, returning to its parent the exit status of the last command executed prior to the end-of-file. The **exit** command simply reads to the end-of-file and returns, setting the exit status to the value of its argument, if any. Thus, a procedure can be terminated "normally" by using **exit 0**.

**Command Grouping—Parentheses/Braces**: There are two methods for grouping commands in the **shell**. As mentioned in "Cd" in subpart "I. Changing the State of the Shell and the *.profile File*", parentheses



( ) cause the **shell** to spawn a *subshell* that reads the enclosed commands. Both the right and left parentheses are recognized *wherever* they appear in a command line. The left and right parentheses can appear as literal parentheses *only* by being quoted. For example, if you type **garble(stuff)**, the **shell** interprets this as four separate words: **garble**, **(**, **stuff**, and **)**.

This subshell capability is useful if one wishes to perform some operations without affecting the values of variables in the current **shell** or to temporarily change directory and execute some commands in the new directory without having to explicitly return to the current directory. The current environment is passed to the subshell and variables that are **exported** in the current **shell** are also **exported** in the subshell. Thus:

```
current=`pwd`; cd /usr/docs/sh_tut;
nohup mm -Tlp sc_? !lpr& cd $current
```

and

```
(cd /usr/docs/sh_tut; nohup mm -Tlp sc_? !lpr&)
```

accomplish the same result. Both examples are used to print a copy of a document on the line printer. However, the second example automatically puts you back in your original working directory. In the second example above, blanks or new-lines surrounding the parentheses are allowed but not necessary. The **shell** will prompt with **\$PS2** is expected. See also the example in "Cd" in subpart "I. Changing the State of the Shell and the .profile File".

Braces **{ }** may also be used to group commands together. See "User-defined Variables" in subpart "D. Shell Variables" and "G. Conditional Substitution" for other meanings of braces in the **shell**. Both the left and the right brace are recognized *only* if they appear as the first (unquoted) word of a command. The opening brace **{** may be followed by a new-line (in which case the **shell** will prompt for more input). Unlike the case involving parentheses, no subshell is spawned for braces. The enclosed commands are simply read by the **shell**. The braces are convenient when you wish to use the (sequential) output of several commands as input to one command. See the last example in "D. Control Commands".

The exit status of a set of commands grouped by either parentheses or braces is the exit status of the last enclosed executed command.

**I/O Redirection and Control Commands:** The **shell** normally does not *fork* when it recognizes the *control* commands (other than parentheses) described above. However, each command in a pipeline is run as a separate process in order to direct input (output) to (from) each command. Also, when redirection of input/output is specified explicitly for a control command, a separate process is spawned to execute that command. Thus, when **if**, **while**, **until**, **case**, or **for** is used in a pipeline consisting of more than one command, the **shell** **forks** and a subshell runs the control command. This has certain implications. The most noticeable one is that *any changes made to variables within the control command are not effective once that control command finishes* (similar to the effect of using parentheses to group commands). The control commands run slightly slower when redirection is specified.

*Beginners should skip to "E. Special Shell Commands" on first reading.*

**In-line Input Documents:** Upon seeing a command line of the form:

```
command << eofstring
```

where *eofstring* is any arbitrary string, the **shell** will take the subsequent lines as the standard input of *command* until a line is read consisting only of *eofstring* (possibly preceded by one or more tab characters). By appending a minus (**-**) to **<<**, leading tab characters are deleted from each line of the input document before the **shell** passes the line to *command*.



The **shell** creates a temporary file containing the input document and performs variable and command substitution ("Command Substitution" in subpart "D. Shell Variables") on its contents before passing it to the command. Pattern matching on file names is performed on the arguments of command lines in command substitutions. In order to prohibit all substitutions, one may quote any character of *eofstring*:

```
command << \eofstring
```

Typically *eofstring* consists of a single character like **!** which is often used for this purpose.

The in-line input document feature is especially useful for small amounts of input data (e.g., an editor "script"), where it is more convenient to place the data in the **shell** procedure than to keep it in a separate file. For instance, one could type:

```
cat <<- xyz
      This message will be printed on the
      terminal with leading tabs removed.
xyz
```

This in-line input document feature is most useful in **shell** procedures. See **edfind**, **edlast**, and **mmt** in part "EXAMPLES OF SHELL PROCEDURES". Note that in-line input documents may *not* appear within grave accents. This is an implementation bug that may be changed in the future.

**Transfer to Another File and Back via Dot (.):** A command line of the form

```
. proc
```

causes the **shell** to read commands from *proc* without spawning a new process. Changes made to variables in *proc* are in effect after the *dot* command finishes. This is thus a good way to gather a number of **shell** variable initializations into one file. Note that an **exit** command in a file executed in this manner will cause an exit from your current **shell**. If you are at login level, you will be logged out.

**Interrupt Handling—"trap":** As noted in "B. UNIX System Processes", a program may choose to *catch* an interrupt from the terminal, *ignore* it completely, or be terminated by it. Shell procedures can use the **trap** command to obtain the same effects.

```
trap arg signal-list
```

is the form of the **trap** command, where *arg* is a string to be interpreted as a command list and *signal-list* consists of one or more signal numbers [as described in **signal(2)**]. The commands in *arg* are scanned at least once when the **shell** first encounters the **trap** command. Because of this, it is usually wise to use single rather than double quotes to surround these commands. The single quotes inhibit immediate command and variable substitution. This becomes important, for instance, when one wishes to remove temporary files and the names of those files have not yet been determined when the **trap** command is first read by the **shell**. The following procedure will print the name of the current directory on the file **errdirect** when it is interrupted, thus giving the user information as to how much of the job was done:

```
trap 'echo `pwd` >errdirect' 2 3 15
for i in /bin /usr/bin /usr/gas/bin
do
    cd $i
    commands to be executed in directory $i here
done
```

while the same procedure with double (rather than single) quotes **trap "echo`pwd`>errdirect" 2 3 15** will, instead, print the name of the directory from which the procedure was executed.



Signal 11 (SEGMENTATION VIOLATION) may never be trapped because the **shell** itself needs to catch it to deal with memory allocation. Zero is not a UNIX system signal but is effectively interpreted by the **trap** command as a signal generated by exiting from a **shell** (either via an **exit** command or by "falling through" the end of a procedure). If *arg* is not specified, then the action taken upon receipt of any of the signals in *signal-list* is reset to the default system action. If *arg* is an explicit null string (" or " "), then the signals in *signal-list* are ignored by the **shell**.

The most frequent use of **trap** is to assure removal of temporary files upon termination of a procedure. The second example of "Predefined Special Variables" in subpart "D. Shell Variables" would be written more typically as follows:

```
temp=$HOME/temp/$$
trap 'rm $temp; trap 0; exit' 0 1 2 3 15
ls > $temp
    commands, some of which use $temp, go here
```

In this example whenever signals 1 (HANGUP), 2 (INTERRUPT), 3 (QUIT), or 15 (SOFTWARE TERMINATION) are received by the shell procedure or whenever the **shell** procedure is about to exit, the commands enclosed between the single quotes will be executed. The **exit** command must be included or else the **shell** continues reading commands where it left off when the signal was received. The **trap 0** turns off the original trap on exits from the **shell** so that the **exit** command does not reactivate the execution of the trap commands.

Sometimes it is useful to take advantage of the fact that the **shell** continues reading commands after executing the trap commands. The following procedure takes each directory in the current directory, changes to it, prompts with its name, and executes commands typed at the terminal until an end-of-file (*control-d*) or an interrupt is received. An end-of-file causes the **read** command to return a nonzero exit status, thus terminating the **while** loop and restarting the cycle for the next directory. The entire procedure is terminated if interrupted when waiting for input; but during the execution of a command, an interrupt terminates *only* that command:

```
dir=`pwd`
for i in *
do
    if test -d $dir/$i
    then
        cd $dir/$i
        while echo " $i:"
        do
            trap exit 2
            read x
        done
        trap : 2      # ignore interrupts
        eval $x
    fi
done
```

Several **traps** may be in effect at the same time. If multiple signals are received simultaneously, they are serviced in ascending order. To check what traps are currently set, type:

```
trap
```

It is important to understand some things about the way in which the **shell** implements the **trap** command in order not to be surprised. When a signal (other than 11) is received by the **shell**, it is passed on to whatever



child processes are currently executing. When those (synchronous) processes terminate, normally or abnormally, the **shell** then polls any traps that happen to be set and executes the appropriate **trap** commands. This process is straightforward except in the case of traps set at the command (outermost or login) level. In this case, it is possible that no child process is running, so the **shell** waits for the termination of the first process spawned after the signal is received before it polls the traps.

For internal commands, the **shell** normally polls traps on completion of the command. An exception to this rule is made for the **read** command, for which traps are serviced immediately, so that **read** can be interrupted while waiting for input.

### E. Special Shell Commands

There are several special commands that are *internal* to the shell (some of which have already been mentioned). These commands should be used in preference to other UNIX system commands whenever possible because they are faster and more efficient. The **shell** does not fork to execute these commands, so no additional processes are spawned. The trade-off for this efficiency is that redirection of input/output is not allowed for most of these special commands.

Several of the special commands have already been described in "D. Control Commands" because they affect the flow of control. They are **break**, **continue**, **exit**, dot (**.**), and **trap**. The **set** command described in "Positional Parameters" in subpart "D. Shell Variables" and "Execution Flags—**set**" in subpart "I. Changing the State of the Shell and the *.profile* File" is also a special command. Descriptions of the remaining special commands [see **sh**(1)] are given here:

- :** The **null** command does nothing. The exit status is zero (*true*). *Beware*: any arguments to the null command are parsed for syntactic correctness; when in doubt, quote such arguments. Parameter substitution takes place just as in other commands.
- cd arg** Make *arg* the current directory. If *arg* does not begin with **/**, **./**, or **../**, **cd** uses the *CDPATH* shell variable ("User-defined Variables" in subpart "D. Shell Variables") to locate a parent directory that contains the directory *arg*. If *arg* is not a directory or the user is not authorized to access it, a nonzero exit status is returned. Specifying **cd** with no *arg* is equivalent to typing **cd \$HOME**.
- exec arg ...** If *arg* is a command, then the **shell** executes it without forking. No new process is created. Input/output redirection arguments *are* allowed on the command line. If *only* input/output redirection arguments appear, then the input/output of the **shell** itself is modified accordingly. See **merge** in part "Examples of Shell Procedures" for an example of this use of **exec**.
- newgrp arg ...** The **newgrp**(1) command is executed replacing the **shell**. The **newgrp** command in turn spawns a new **shell**. *Beware*: Only variables in the environment will be known in the shell that is spawned by the **newgrp** command. Any variables that were **exported** will no longer be marked as such.
- read var ...** One line (up to a new-line) is read from standard input and the first word is assigned to the first variable, the second word to the second variable, etc. All leftover words are assigned to the *last* variable. The exit status of **read** is zero unless an end-of-file is read.
- readonly var ...** The specified variables are made **readonly** so that no subsequent assignments may be made to them. If no arguments are given, a list of all **readonly** and of all **exported** variables is given.
- test** A conditional expression is evaluated. More details are given in "A. Conditional Evaluation—**test**".



|                         |                                                                                                                                                                                                                                                                                                                                                                   |
|-------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>times</b>            | The accumulated user and system times for processes run from the current <b>shell</b> are printed.                                                                                                                                                                                                                                                                |
| <b>umask <i>nnn</i></b> | The user file creation mask is set to <i>nnn</i> . See <b>umask(2)</b> for details. If <i>nnn</i> is omitted, then the current value of the mask is printed.                                                                                                                                                                                                      |
| <b>ulimit <i>n</i></b>  | This command imposes a limit of <i>n</i> blocks on the size of files written by the <b>shell</b> and its child processes (files of any size may be read). If <i>n</i> is omitted, the current value of this limit is printed. The default value for <i>n</i> varies from one installation to another.                                                             |
| <b>wait <i>n</i></b>    | The <b>shell</b> waits for the child process whose process number is <i>n</i> to terminate. The exit status of the <b>wait</b> command is that of the process waited on. If <i>n</i> is omitted or is not a child of the current <b>shell</b> , then <i>all</i> currently active processes are waited for and the return code of the <b>wait</b> command is zero. |

#### F. Creation and Organization of Shell Procedures

A shell procedure can be created in two simple steps:

1. Build an ordinary text file.
2. Change the file's *mode* to make it *executable*.

Changing the *mode* allows a shell procedure to be invoked by *proc args* rather than by **sh proc args**. The second step may be omitted for a procedure to be used once or twice and then discarded but is recommended for longer-lived ones. Here is the entire input needed to set up a simple procedure (the executable part of **draft** in part "Examples of Shell Procedures"):

```
ed
a
nroff -rC3 -T450-12 -cm $*
.
w draft
q
chmod +x draft
```

It may then be invoked as **draft file1 file2**. Note that **shell** procedures must always be at least readable so that the **shell** itself can read commands from the file.

If **draft** were thus created in a directory whose name appears in the user's *PATH* variable, the user could change working directories and still invoke the **draft** command.

Shell procedures may be created dynamically. A procedure may generate a file of commands, invoke another instance of the **shell** to execute that file, and then remove it. An alternate approach is that of using the *dot* command (.) to make the current **shell** read commands from the new file, allowing use of existing **shell** variables, and avoiding the spawning of an additional process for another **shell**.

Many users prefer to write **shell** procedures instead of C programs. First, it is easy to create and maintain a **shell** procedure because it is only a file of ordinary text. Second, it has no corresponding object program that must be generated and maintained. Third, it is easy to create a procedure on the fly, use it a few times, and then remove it. Finally, because **shell** procedures are usually short in length, written in a high-level programming language, and kept only in their source-language form, they are generally easy to find, understand, and modify.

By convention, directories that contain only commands and/or shell procedures are usually named *bin*. Most groups of users sharing common interests have one or more *bin* directories set up to hold common procedures.



Some users have their *PATH* variable list several such directories. Although you can have a number of such directories, it is unwise to go overboard. It may become difficult to keep track of your environment, and efficiency may suffer (see "C. Efficient Organization").

#### G. More about Execution Flags

There are several execution flags available in the *shell* that can be useful in *shell* procedures:

- e           The *shell* will exit immediately if any command that it executes exits with a nonzero exit status.
- u           When this flag is set, the *shell* treats the use of an unset variable as an error. This flag can be used to perform a global check on variables.
- t           The *shell* exits after reading and executing the commands on the remainder of the current input line.
- n           This is a *don't execute* flag. On occasion, one may want to check a procedure for syntax errors but not to execute the commands in the procedure. Writing *set -nv* at the beginning of the file will accomplish this.
- k           All arguments of the form *variable=value* are treated as keyword parameters. When this flag is *not* set, only such arguments that appear *before* the command name are treated as keyword parameters.

#### MISCELLANEOUS SUPPORTING COMMANDS AND FEATURES

*Shell* procedures can make use of any UNIX system command. The commands described in this part are either used especially frequently in *shell* procedures or are explicitly designed for such use. More detailed descriptions of each of these commands can be found in Section 1 of the UNIX System User's Manual.

##### A. Conditional Evaluation—"test"

The *test*(1) command evaluates the expression specified by its arguments and, if the expression is true, returns a zero exit status. Otherwise, a nonzero (false) exit status is returned. The *test* command also returns a nonzero exit status if it has no arguments. Often it is convenient to use the *test* command as the first command in the *command list* following an *if* or a *while*. Shell variables used in *test* expressions should be enclosed in double quotes if there is any chance of their being null or not set.

On some UNIX systems, the square brackets (*[]*) may be used as an alias for *test*; e.g., [*expression*] has the same effect as *test expression*.

The following is a partial list of the primaries that can be used to construct a conditional expression:

- r *file*           *true* if the named file exists and is readable by the user.
- w *file*           *true* if the named file exists and is writable by the user.
- x *file*           *true* if the named file exists and is executable by the user.
- s *file*           *true* if the named file exists and has a size greater than zero.
- d *file*           *true* if the named file exists and is a directory.
- f *file*           *true* if the named file exists and is an ordinary file.



|                        |                                                                                                                                                                                                                              |
|------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-p file</code>   | <i>true</i> if the named file exists and is a named pipe ( <i>fifo</i> ).                                                                                                                                                    |
| <code>-z s1</code>     | <i>true</i> if the length of string "s1" is zero.                                                                                                                                                                            |
| <code>-n s1</code>     | <i>true</i> if the length of the string "s1" is nonzero.                                                                                                                                                                     |
| <code>-t fildes</code> | <i>true</i> if the open file whose file descriptor number is <i>fildes</i> is associated with a terminal device. If <i>fildes</i> is not specified, file descriptor 1 is used by default.                                    |
| <code>s1 = s2</code>   | <i>true</i> if strings "s1" and "s2" are identical.                                                                                                                                                                          |
| <code>s1 != s2</code>  | <i>true</i> if strings "s1" and "s2" are <i>not</i> identical.                                                                                                                                                               |
| <code>s1</code>        | <i>true</i> if "s1" is <i>not</i> the null string.                                                                                                                                                                           |
| <code>n1 -eq n2</code> | <i>true</i> if the integers <i>n1</i> and <i>n2</i> are algebraically equal. Other algebraic comparisons are indicated by <code>-ne</code> , <code>-gt</code> , <code>-ge</code> , <code>-lt</code> , and <code>-le</code> . |

These primaries may be combined with the following operators:

|                       |                                                                                                                                                                           |
|-----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>!</code>        | unary negation operator.                                                                                                                                                  |
| <code>-a</code>       | binary logical <i>and</i> operator.                                                                                                                                       |
| <code>-o</code>       | binary logical <i>or</i> operator. The <code>-o</code> has lower precedence than <code>-a</code> .                                                                        |
| <code>( expr )</code> | parentheses for grouping; they must be escaped to remove their significance to the <b>shell</b> . When parentheses are absent the evaluation proceeds from left to right. |

Note that all primaries, operators, file names, etc. are separate arguments to **test**.

#### B. Reading a Line—"line"

The `line(1)` command takes one line from standard input and prints it on standard output. This is useful when you need to read a line from a file or capture the line in a variable. The functions of **line** and of the **read** command that is internal to the **shell** differ in that input/output redirection is possible only with **line**. If the user does not require input/output redirection, **read** is faster and more efficient. An example of a usage of **line** for which **read** would not suffice is:

```
firstline='line < somefile'
```

#### C. Simple Output—"echo"

The `echo(1)` command, invoked as `echo [ arg... ]`, copies its arguments to the standard output, each followed by a single space except for the last argument which is normally followed by a new-line. Often, **echo** is used to prompt the user for input to issue diagnostics in **shell** procedures or to add a few lines to an output stream in the middle of a pipeline. Another use is to verify the argument list generation process before issuing a command that does something drastic. The command **ls** is often replaced by `echo *` because the latter is faster and prints fewer lines of output.

The **echo** command recognizes several escape sequences. A `\n` yields a new-line character. A `\c` removes the new-line from the end of the echoed line. The following prompts the user, allowing one to type on the same line as the prompt:

```
echo 'enter name:\c'
read name
```



The `echo` command also recognizes octal escape sequences for *all* characters whether printable or not. An `echo "\007"` typed at a terminal will cause the bell on that terminal to ring.

#### D. Expression Evaluation—"expr"

The `expr(1)` command provides arithmetic and logical operations on integers and some pattern matching facilities on its arguments. It evaluates a single expression and writes the result on the standard output. The `expr` command can be used inside grave accents to set a variable. Typical examples are:

```
#          increment $a
a=`expr $a + 1`
#          put third through last characters of
#          $1 into substring
substring=`expr " $1" : '.*\(.*\)'`
#          obtain length of $1
c=`expr " $1" : '.*'`
```

The most common uses of `expr` are in counting iterations of a loop and in using its pattern matching capability to pick apart strings. See `expr(1)` for more details.

#### E. "true" and "false"

The `true(1)` and `false` [see `true(1)`] commands perform the obvious functions of exiting with zero and non-zero exit status, respectively. The `true` command is often used to implement an unconditional loop.

#### F. Input/Output Redirection Using File Descriptors

Beginners should skip this subpart on first reading. A command occasionally directs output to some file associated with a file descriptor other than 1 or 2 (see "Diagnostic and Other Outputs" in subpart "F. Redirection of Input and Standard Output"). In languages such as C, one can associate output with *any* file descriptor by using the `write(2)` system call. The `shell` provides its own mechanism for creating an output file associated with a particular file descriptor. By typing

```
fd1>&fd2
```

where `fd1` and `fd2` are valid file descriptors, one can direct output that would normally be associated with file descriptor `fd1` onto the file associated with `fd2`. The default value for `fd1` and `fd2` is 1. If, at execution time, no file is associated with `fd2`, then the redirection is void. The most common use of this mechanism is that of directing standard error output to the same file as standard output. This is accomplished by typing

```
command 2>&1
```

If one wanted to redirect both standard output and standard error output to the same file, one would type

```
command 1> file 2>&1
```

*The order here is significant.* First, file descriptor 1 is associated with *file*. Then file descriptor 2 is associated with the same file that is *currently* associated with file descriptor 1. If the order of the redirections were reversed, standard error output would go to the terminal, and standard output would go to *file* because at the time of the error output redirection file descriptor 1 still would have been associated with the terminal.

This mechanism can also be generalized to the redirection of standard *input*. One could type

```
fda<&fdb
```



to cause both file descriptors *fda* and *fdb* to be associated with the same input file. If *fda* or *fdb* is not specified, file descriptor 0 is assumed. Such input redirection is useful for commands that use two or more input sources. Another use of this notation is for sequential reading and processing of a file. See *merge* in part "EXAMPLES OF SHELL PROCEDURES" for an example of use of this feature.

#### G. Conditional Substitution

Normally, the *shell* replaces occurrences of *\$variable* by the string value assigned to *variable*, if any. However, there exists a special notation to allow conditional substitution depending upon whether the variable is set and/or not null. By definition, a variable is *set* if it has *ever* been assigned a value. The value of a variable can be the null string which may be assigned to a variable in any one of the following ways:

```
A=
bcd=""
Ef_g=' '
set ' ' ""
```

The first three of these examples assign the null string to each of the corresponding *shell variables*. The last example sets the first and second *positional parameters* to the null string and *unsets* all other positional parameters.

The following conditional expressions depend upon whether a variable is *set and not null*. (Note that, in these expressions, *variable* refers to either a digit or a variable name and the meaning of braces differs from that described in "User-defined Variables" in subpart "D. Shell Variables" and "Command Grouping—Parentheses and Braces" in subpart "D. Control Commands".)

##### *\${variable :- string}*

If *variable* is set and is non-null, then substitute the value *\$variable* in place of this expression. Otherwise, replace the expression with *string*. Note that the value of *variable* is *not* changed by the evaluation of this expression.

##### *\${variable := string}*

If *variable* is set and is non-null, then substitute the value *\$variable* in place of this expression. Otherwise, set *variable* to *string*, and then substitute the value *\$variable* in place of this expression. Positional parameters may not be assigned values in this fashion.

##### *\${variable :? string}*

If *variable* is set and is non-null, then substitute the value of *variable* for the expression. Otherwise, print a message of the form

```
variable:      string
```

and exit from the current *shell*. (If the *shell* is the login *shell*, it is not exited.) If *string* is omitted in this form, then the message

```
variable:      parameter null or not set
```

is printed instead.

##### *\${variable :+ string}*

If *variable* is set and is non-null, then substitute *string* for this expression; otherwise, substitute the null string. Note that the value of *variable* is not altered by the evaluation of this expression.



These expressions may also be used without the colon (:), in which case the **shell** does *not* check whether *variable* is null or not. It only checks whether *variable* has *ever* been set.

The two examples below illustrate the use of this facility:

1. If *PATH* has ever been set and is not null, then keep its current value. Otherwise, set it to the string *:/bin:/usr/bin*. Note that one needs an explicit assignment to set *PATH* in this form:

```
PATH=${PATH:-'/bin:/usr/bin'}
```

2. If *HOME* is set and is not null, then change directory to it; otherwise, set it to the given value and change directory to it. Note that *HOME* is automatically assigned a value in this case:

```
cd ${HOME:='/usr/gas'}
```

#### H. Invocation Flags

There are four flags that may be specified on the command line invoking the **shell**. These flags may *not* be turned on via the **set** command:

- i      If this flag is specified or if the **shell**'s input and output are both attached to a terminal, the **shell** is *interactive*. In such a **shell**, INTERRUPT (signal 2) is caught and ignored, while QUIT (signal 3) and SOFTWARE TERMINATION (signal 15) are ignored.
- s      If this flag is specified or if no input/output redirection arguments are given, the **shell** reads commands from standard input. Shell output is written to file descriptor 2. The **shell** you get upon logging into the system effectively has the **-s** flag turned on.
- c      When this flag is turned on, the **shell** reads commands from the first string following the flag. Remaining arguments are ignored. Double quotes should be used to enclose a multiword string in order to allow for variable substitution.
- r      When this flag is specified on invocation, then the *restricted shell* is invoked. This is a version of the **shell** in which certain actions are disallowed. In particular, the **cd** command produces an error message, and the user cannot set *PATH*. See **sh(1)** for a more detailed description.

#### EXAMPLES OF SHELL PROCEDURES

Some examples in this subpart are quite difficult for beginners. For ease of reference, the examples are arranged alphabetically by name, rather than by degree of difficulty.

##### *coppairs*

```
#          usage: coppairs file1 file2 ...
#          copy file1 to file2, file3 to file4, ...
while test "$2" != ""
do
    cp $1 $2
    shift; shift
done
if test "$1" != ""
then
    echo "$0: odd number of arguments"
fi
```



**Note:** This procedure illustrates the use of a **while** loop to process a list of positional parameters that are somehow related to one another. Here a **while** loop is much better than a **for** loop because you can adjust the positional parameters via **shift** to handle related arguments.

### *copyto*

```
#          usage: copyto dir file ...
#          copy argument files to 'dir', making sure that at least
#          two arguments exist and that 'dir' is a directory
if test $# -lt 2
then
    echo " $0: usage: copyto directory file ..."
elif test ! -d $1
then
    echo " $0: $1 is not a directory";
else
    dir=$1; shift
    for eachfile
    do
        cp $eachfile $dir
    done
fi
```

**Note:** This procedure uses an **if** command with two tests in order to screen out improper usage. The **for** loop at the end of the procedure loops over all of the arguments to **copyto** but the first. The original **\$1** is shifted off.

### *distinct*

```
#          usage: distinct
#          reads standard input and reports list of alphanumeric strings
#          that differ only in case, giving lowercase form of each
tr -cs '[A-Z][a-z][0-9]' '\012*' | sort -u |
tr '[A-Z]' '[a-z]' | sort | uniq -d
```

**Note:** This procedure is an example of the kind of process that is created by the left-to-right construction of a long pipeline. It may not be immediately obvious how this works. [See **tr(1)**, **sort(1)**, and **uniq(1)** if you are completely unfamiliar with these commands.] The **tr** translates all characters except letters and digits into new-line characters and then squeezes out repeated new-line characters. This leaves each string (in this case, any contiguous sequence of letters and digits) on a separate line. The **sort** command sorts the lines and emits only one line from any sequence of one or more repeated lines. The next **tr** converts everything to lowercase, so that identifiers differing only in case become identical. The output is sorted again to bring such duplicates together. The **uniq -d** prints (once) only those lines that occur more than once yielding the desired list.

The process of building such a pipeline uses the fact that pipes and files can usually be interchanged. The two lines below are equivalent assuming that sufficient disk space is available:

```
cmd1 | cmd2 | cmd3
cmd1 > temp1; < temp1 cmd2 > temp2; < temp2 cmd3; rm temp[12]
```



Starting with a file of test data on the standard input and working from left to right, each command is executed taking its input from the previous file and putting its output in the next file. The final output is then examined to make sure that it contains the expected result. The goal is to create a series of transformations that will convert the input to the desired output. As an exercise, try to mimic **distinct** with such a step-by-step process using a file of test data containing:

```
ABC:DEF/DEF
ABC1 ABC
Abc,abc
```

Although pipelines can give a concise notation for complex processes exercise some restraint lest you succumb to the "one-line syndrome" sometimes found among users of especially concise languages. This syndrome often yields incomprehensible code.

### **draft**

```
#      usage: draft file(s)
#      prints the draft (-rC3) of a document on a DASI 450
#      terminal in 12-pitch using memorandum macros (MM).
nroff -rC3 -T450-12 -cm $*
```

**Note:** Users often write this kind of procedure for convenience in dealing with commands that require the use of many distinct flags that cannot be given default values that are reasonable for all (or even most) users.

### **edfind**

```
#      usage: edfind file arg
#      find the last occurrence in 'file' of a line whose
#      beginning matches 'arg', then print 3 lines (the one
#      before, the line itself, and the one after)
ed - $1 << !
H
?^$2?;-,+p
!
```

**Note:** This procedure illustrates the practice of using editor (**ed**) in-line input scripts into which the shell can substitute the values of variables. It is a good idea to turn on the **H** option of **ed** when embedding an **ed** script in a shell procedure [see **ed(1)**].

### **edlast**

```
#      usage: edlast file
#      prints the last line of file, then deletes that line
ed - $1 << - \eof      # no variable substitutions in "ed" script
H
$P
$d
w
q
eof
echo Done.
```



**Note:** This procedure contains an in-line input document or script (see "In-line Input Documents" in subpart "D. Control Commands"); it also illustrates the effect of inhibiting substitution by escaping a character in the *eofstring* (here, *eof*) of the input redirection. If this had not been done, *\$p* and *\$d* would have been treated as *shell* variables.

### *fsplit*

```
#          usage: fsplit file1 file2
#          read standard input and divide it into three parts:
#          append any line containing at least one letter
#          to file1, any line containing at least one digit
#          but no letters to file2, and throw the rest away
total=0 lost=0
while read next
do
    total="`expr $total + 1`"
    case "$next" in
        *[A-Za-z]*)
            echo "$next" >> $1 ;;
        *[0-9]*)
            echo "$next" >> $2 ;;
        *)
            lost="`expr $lost + 1`"
    esac
done
echo "$total lines read, $lost thrown away"
```

**Note:** In this procedure, each iteration of the *while* loop reads a line from the input and analyzes it. The loop terminates only when *read* encounters an end-of-file.

Do not use the *shell* to read a line at a time unless you must—it can be grotesquely slow (see "Number of Processes Generated" in subpart "B. Approximate Measures of Resource Consumption").

### *initvars*

```
#          usage: . initvars
#          use carriage return to indicate "no change"
echo "initializations? \c"
read response
if test "$response" = y
then
    echo "PS1=\c"; read temp
    PS1=${temp:-$PS1}
    echo "PS2=\c"; read temp
    PS2=${temp:-$PS2}
    echo "PATH=\c"; read temp
    PATH=${temp:-$PATH}
    echo "TERM=\c"; read temp
    TERM=${temp:-$TERM}
fi
```

**Note:** This procedure would be invoked by a user at the terminal or as part of a *.profile* file. The assignments are effective even when the procedure is finished because the *dot* command is used to invoke it. To



better understand the `dot` command invoke `initvars` as indicated above and check the values of `PS1`, `PS2`, `PATH`, and `TERM`; then make `initvars` executable, type `initvars`, assigning different values to the three variables, and check again the values of these three shell variables after `initvars` terminates. It is assumed that `PS1`, `PS2`, `PATH`, and `TERM` have been exported, presumably by your `.profile` (see "The `.profile` File" in subpart "I. Changing the State of the Shell and the `.profile` File" and "A. A Command's Environment").

### **merge**

```
#      usage:  merge src1 src2 [ dest ]
#      merge two files, every other line.
#      the first argument starts off the merge,
#      excess lines of the longer file are appended to
#      the end of the resultant file
exec 4<$1 5<$2
dest=${3-$1.m}          # default destination file is named $1.m
while true
do
    # alternate reading from the files;
    # 'more' represents the file descriptor
    # of the longer file
    line <&4 >>$dest  ||{ more=5; break ;}
    line <&5 >>$dest  ||{ more=4; break ;}
done
    # delete the last line of destination
    # file, because it is blank.

ed - $dest <<\eof
H
$d
w
q
eof
while line <&$more >> $dest
do ;; done          # read the remainder of the longer
                    # file—the body of the 'while' loop
                    # does nothing; the work of the loop
                    # is done in the command list following
                    # 'while'
```

**Note:** This procedure illustrates a technique for reading sequential lines from a file or files without creating any subshells to do so. When the file descriptor is used to access a file, the effect is that of opening the file and moving a file pointer along until the end of the file is read. If the input redirections used `src1` and `src2` explicitly rather than the associated file descriptors, this procedure would never terminate because the *first* line of each file would be read over and over again.

### **mkfiles**

```
#      usage: mkfiles pref [ quantity ]
#      makes 'quantity' (default = 5) files, named pref1, pref2, ...
quantity=${2-5}
i=1
while test "$i" -le "$quantity"
do
    > $1$i
    i=`expr $i + 1`
done
```



**Note:** This procedure uses input/output redirection to create zero-length files. The `expr` command is used for counting iterations of the `while` loop. Compare this procedure with procedure `null` below.

`mmt`

```
if test "$#" = 0; then cat <<\!
```

```
Usage: " mmt [ options ] files" where " options" are:
```

```
-a          => output to terminal
-e          => preprocess input with eqn
-t          => preprocess input with tbl
-Tst        => output to STARE phototypesetter
             manufactured by Honeywell
-T4014      => output to 4014 manufactured by Tektronix
-Tvp        => output to printer manufactured by Versatec
-           => use instead of " files" when mmt used
             inside a pipeline.
```

```
Other options as required by TROFF and the MM macros.
```

```
!
```

```
    exit 1
```

```
fi
```

```
PATH='/bin:/usr/bin'; O='-g'; o='! gcat -ph';
```

```
#           Assumes typesetter is accessed via gcat(1)
```

```
#           If typesetter is on-line, use O=''; o=''
```

```
while test -n "$1" -a ! -r "$1"
```

```
do
```

```
    case "$1" in
```

```
        -a)          O='-a';
```

```
        o='';
```

```
        -Tst)        O='-g';
```

```
        o='!gcat -st';
```

```
#
```

```
    Above line for STARE only
```

```
        -T4014)      O='-t';
```

```
        o='!tc';
```

```
        -Tvp)        O='-t';
```

```
        o='!vpr -t';
```

```
        -e)          e='eqn';
```

```
        -t)          f='tbl';
```

```
        -)           break;
```

```
        *)           a="$a $1";
```

```
    esac
```

```
    shift
```

```
done
```

```
if test -z "$1"
```

```
then
```

```
    echo 'mmt: no input file'
```

```
    exit 1
```

```
fi
```

```
if test "$O" = '-g'
```

```
then
```

```
    x="-f$1"
```

```
fi
```

```
d="$*"
```

```
if test "$d" = '-'
```

```
then
```

```
    shift
```

```
    x=''
```

```
    d=''
```



```

fi
if test -n "$f"
then
    f="tbl $*"
    d=' '
fi
if test -n "$e"
then
    if test -n "$f"
    then e='eqn !'
    else e="eqn $*"
    d=''
fi
fi
eval "$f $e troff $O -cm $a $d $O $x";      exit 0

```

**Note:** This is a slightly simplified version of an actual UNIX system command (although this is *not* the version included in UNIX system Release 4.0). It uses many of the features available in the **shell**. If you can follow through it without getting lost, you have a good understanding of **shell** programming. Pay particular attention to the process of building a command line from **shell** variables and then using **eval** to execute it.

### **null**

```

#          usage: null file
#          create each of the named files as an empty file
for eachfile
do
    > $eachfile
done

```

**Note:** This procedure uses the fact that output redirection creates the (empty) output file if that file does not already exist. Compare this procedure with procedure **mkfiles** above.

### **phone**

```

#          usage: phone initials
#          prints the phone number(s) of person with given initials
echo 'inits      ext      home'
grep "^$1" <<\!
abc          1234      999-2345
def          2234      583-2245
ghi          3342      988-1010
xyz          4567      555-1234
!

```

**Note:** This procedure is an example of using an in-line input document or *script* to maintain a *small* data base.

### **writemail**

```

#          usage: writemail message user
#          if user is logged in, write message on terminal;
#          otherwise, mail it to user
echo "$1" | { write "$2" || mail "$2" ;}

```



**Note:** This procedure illustrates command grouping. The message specified by **\$1** is piped to the **write** command and, if **write** fails, to the **mail** command.

## EFFECTIVE AND EFFICIENT SHELL PROGRAMMING

### A. Overall Approach

This subpart outlines strategies for writing *efficient* shell procedures, i.e., ones that do not waste resources unreasonably in accomplishing their purposes. The primary reason for choosing the **shell** procedure as the implementation method is to achieve a desired result at a minimum *human* cost. Emphasis should *always* be placed on simplicity, clarity, and readability; but efficiency can also be gained through awareness of a few design strategies. In many cases, an effective redesign of an existing procedure improves its efficiency by reducing its size and often increases its comprehensibility. In any case, one should not worry about optimizing **shell** procedures unless they are intolerably slow or are known to consume a lot of resources.

The same kind of iteration cycle should be applied to **shell** procedures as to other programs—write code, measure it, and optimize only the *few* important parts. The user should become familiar with the **time** command which can be used to measure both entire procedures and parts thereof. Its use is strongly recommended; human intuition is notoriously unreliable when used to estimate timings of programs even when the style of programming is a familiar one. Each timing test should be run several times because the results are easily disturbed by, for instance, variations in system load.

### B. Approximate Measures of Resource Consumption

**Number of Processes Generated:** When large numbers of short commands are executed, the actual execution time of the commands may well be dominated by the overhead of creating processes. The procedures that incur significant amounts of such overhead are those that perform much looping and those that generate command sequences to be interpreted by another **shell**.

If you are worried about efficiency, it is important to know which commands are currently built into the **shell** and which are not. Here is the alphabetical list of those that are built-in:

|                 |               |              |                 |               |             |
|-----------------|---------------|--------------|-----------------|---------------|-------------|
| <b>break</b>    | <b>case</b>   | <b>cd</b>    | <b>continue</b> | <b>eva</b>    | <b>exec</b> |
| <b>exit</b>     | <b>export</b> | <b>for</b>   | <b>if</b>       | <b>newgrp</b> | <b>read</b> |
| <b>readonly</b> | <b>set</b>    | <b>shift</b> | <b>test</b>     | <b>times</b>  | <b>trap</b> |
| <b>ulimit</b>   | <b>umask</b>  | <b>until</b> | <b>wait</b>     | <b>while</b>  | .           |
| :               | {...}         |              |                 |               |             |

The (...) command executes as a child process, i.e., the **shell** does a **fork**, but no **exec**. Any command *not* in the above list requires both **fork** and **exec**.

The user should always have at least a vague idea of the number of processes generated by a **shell** procedure. In the bulk of observed procedures, the number of processes spawned (not necessarily simultaneously) can be described by

$$\text{processes} = k * n + c$$

where *k* and *c* are constants, and *n* is the number of procedure arguments, the number of lines in some input file, the number of entries in some directory, or some other obvious quantity. Efficiency improvements are most commonly gained by reducing the value of *k*, sometimes to zero. Any procedure whose complexity measure includes *n*<sup>2</sup> terms or higher powers of *n* is likely to be intolerably expensive.

As an example, here is an analysis of procedure **fsplit** of part "EXAMPLES OF SHELL PROCEDURES". For each iteration of the loop, there is one **expr** plus either an **echo** or another **expr**. One additional **echo**



is executed at the end. If  $n$  is the number of lines of input, the number of processes is  $2*n + 1$ . On the other hand, the number of processes in the following (equivalent) procedure is 12 regardless of the number of lines of input:

```
#          faster fsplit
trap 'rm temp$$; trap 0; exit' 0 1 2 3 15
start1=0 start2=0
b='[A-Za-z]'
cat > temp$$          # read standard input into temp file
                      # save original lengths of $1, $2
if test -s "$1"; then start1='wc -l < $1'; fi
if test -s "$2"; then start2='wc -l < $2'; fi
grep "$b" temp$$ >> $1      # lines with letters onto $1
grep -v "$b" temp$$ | grep '[0-9]' >> $2
                      # lines with only numbers onto $2
total='wc -l < temp$$'
end1='wc -l < $1'
end2='wc -l < $2'
lost='expr $total - \( $end1 - $start1 \) - \( $end2 - $start2 \) '
echo "$total lines read, $lost thrown away"
```

This version is often ten times faster than `fsplit`, and it is even faster for larger input files.

Some types of procedures should *not* be written using the `shell`. For example, if one or more processes are generated for each character in some file, it is a good indication that the procedure should be rewritten in C.

**Note:** Shell procedures should not be used to scan or build files a character at a time.

**Number of Data Bytes Accessed:** It is worthwhile considering any action that reduces the number of bytes read or written. This may be important for those procedures whose time is spent passing data around among a few processes rather than in creating large numbers of short processes. Some filters shrink their output, others usually increase it. It always pays to put the *shrinkers* first when the order is irrelevant. Which of the following is likely to be faster?

```
sort file | grep pattern
grep pattern file | sort
```

**Directory Searches:** Directory searching can consume a great deal of time, especially in those applications that utilize deep directory structures and long pathnames. Judicious use of `cd` can help shorten long pathnames and thus reduce the number of directory searches needed. As an exercise, try the following commands (on a fairly quiet system):

```
time sh -c 'ls -l /usr/bin/* >/dev/null'
time sh -c 'cd /usr/bin; ls -l * >/dev/null'
```

If you do not understand exactly what is going on in these examples, read Section 7 in the UNIX System User's Manual.

### C. Efficient Organization

**Directory-Search Order and PATH Variable:** The `PATH` variable is a convenient mechanism for allowing organization and sharing of procedures. However, it must be used in a sensible fashion; or the result may be a great increase in system overhead that occurs in a subtle, but avoidable, way.

The process of finding a command involves reading every directory included in every pathname that precedes the needed pathname in the current `PATH` variable. As an example, consider the effect of invoking `nroff`



(i.e., `/usr/bin/nroff`) when `$PATH` is `:/bin:/usr/bin`. The sequence of directories read is: `.`, `/`, `/bin`, `/usr`, and `/usr/bin`, i.e., a total of six directories. A long path list assigned to `PATH` can increase this number significantly.

The vast majority of command executions are of commands found in `/bin` and, to a somewhat lesser extent, in `/usr/bin`. Careless `PATH` setup may lead to a great deal of unnecessary searching. The following four examples are ordered from worst to best (but *only* with respect to the efficiency of command searches):

```
:/a1/tf/jtb/bin:/usr/lbin:/bin:/usr/bin
:/bin:/a1/tf/jtb/bin:/usr/lbin:/usr/bin
:/bin:/usr/bin:/a1/tf/jtb/bin:/usr/lbin
/bin::/usr/bin:/a1/tf/jtb/bin:/usr/lbin
```

The first one above should be avoided. The others are acceptable; the choice among them is dictated by the rate of change in the set of commands kept in `/bin` and `/usr/bin`.

A procedure that is expensive because it invokes many short-lived commands may often be speeded up by setting the `PATH` variable inside the procedure such that the fewest possible directories are searched in an optimum order. The `mmt` example in part "EXAMPLES OF SHELL PROCEDURES" does this.

**Setting Up Directories:** It is wise to avoid directories that are larger than necessary. You should be aware of several *magic* sizes. A directory that contains entries for up to 30 files (plus the required `.` and `..`) fits in a single disk block and can be searched very efficiently. One that has up to 286 entries is still a *small* file. Anything larger is usually a disaster when used as a working directory. It is especially important to keep login directories small, preferably one block at most. Note that, as a rule, directories never shrink.

## REFERENCES

- [1]—Bianchi, M. H., and Wood, J. L. A User's Viewpoint on the Programmer's Workbench. Proc. Second Int. Conf. on Software Engineering, pp. 193-99 (Oct. 13-15, 1976).
- [2]—Dolotta, T. A., Haight, R. C., and Mashey, J. R. The Programmer's Workbench. The Bell System Technical Journal, Vol. 57, No. 6, Part 2, pp. 2177-200 (July-Aug. 1978).
- [3]—Dolotta, T. A., and Mashey, J. R. An Introduction to the Programmer's Workbench. Proc. Second Int. Conf. on Software Engineering, pp. 164-68 (Oct. 13-15, 1976).
- [4]—Dolotta, T. A., and Mashey, J. R. Using a Command Language as the Primary Programming Tool. In: Beech, D. (ed.), Command Language Directions (Proc. of the Second IFIP Working Conf. on Command Languages), pp. 35-55. Amsterdam: North Holland (1980).
- [5]—Kernighan, B. W., and Mashey, J. R. The UNIX Programming Environment. COMPUTER, Vol. 14, No. 4, pp. 12-24 (April 1981); an earlier version of this paper was published in Software-Practice & Experience, Vol. 9, No. 1, pp. 1-15 (Jan. 1979).
- [6]—Kernighan, B. W., and Plauger, P. J. Software Tools. Proc. First Nat. Conf. on Software Engineering, pp. 8-13 (Sept. 11-12, 1975).
- [7]—Kernighan, B. W., and Plauger, P. J. Software Tools. Reading, MA: Addison-Wesley (1976).
- [8]—Kernighan, B. W., and Ritchie, D. M. The C Programming Language. Englewood Cliffs, NJ: Prentice-Hall (1978).
- [9]—Ritchie, D. M., and Thompson, K. The UNIX Time-Sharing System. The Bell System Technical Journal, Vol. 57, No. 6, Part 2, pp. 1905-29 (July-Aug. 1978).
- [10]—Snyder, G. A., and Mashey, J. R. UNIX System Documentation Road Map. Bell Laboratories (January 1981).



### 3. THE C PROGRAMMING LANGUAGE

#### INTRODUCTION

This section describes C language as it is implemented and supported on the 3B20S Processor, the PDP\*-11, the VAX\*-11/780, the Honeywell 6000, the IBM System/370, and the Interdata 8/32. Where differences exist, this section concentrates on the PDP-11 but tries to point out implementation-dependent details. With few exceptions, such dependencies follow directly from the properties of the hardware. The various compilers are generally quite compatible. This section contains the following subsections:

- **C LANGUAGE**—A summary of the grammar and rules of the C programming language.
- **LIBRARIES**—Descriptions of functions and declarations that support C language and how to use these functions.
- **THE “cc” COMMAND**—The command used to compile C language programs, assemble assembly language programs, and produce executable programs is briefly described in terms of usage.
- **A C PROGRAM CHECKER—“lint”**—A program that attempts to detect bugs in C programs during compilation.
- **A SYMBOLIC DEBUGGER—“sdb”**—A symbolic debugging program that is used to debug compiled C language programs.

Throughout this section, each reference of the form **name(1M)**, **name(7)**, or **name(8)** refers to entries in the UNIX System Administrator's Manual. All other references to entries of the form **name(N)**, where “N” is a number (1 through 6) possibly followed by a letter, refer to entry **name** in section N of the UNIX System User's Manual.

\*Trademarks of the Digital Equipment Corporation.



## C LANGUAGE

## LEXICAL CONVENTIONS

There are six classes of tokens—identifiers, keywords, constants, strings, operators, and other separators. Blanks, tabs, new-lines, and comments (collectively, “white space”) as described below are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

## A. Comments

The characters `/*` introduce a comment which terminates with the characters `*/`. Comments do not nest.

## B. Identifiers (Names)

An identifier is a sequence of letters and digits. The first character must be a letter. The underscore (`_`) counts as a letter. Uppercase and lowercase letters are different. No more than the first eight characters are significant, although more may be used. External identifiers, which are used by various assemblers and loaders, are more restricted:

|                |                       |
|----------------|-----------------------|
| PDP-11         | 7 characters, 2 cases |
| VAX-11         | 7 characters, 2 cases |
| Honeywell 6000 | 6 characters, 1 case  |
| IBM 360/370    | 7 characters, 1 case  |
| Interdata 8/32 | 8 characters, 2 cases |
| WEC0 3B20      | 8 characters, 2 cases |

## C. Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

|                       |                     |                    |                       |                       |
|-----------------------|---------------------|--------------------|-----------------------|-----------------------|
| <code>auto</code>     | <code>do</code>     | <code>float</code> | <code>register</code> | <code>switch</code>   |
| <code>break</code>    | <code>double</code> | <code>for</code>   | <code>return</code>   | <code>typedef</code>  |
| <code>case</code>     | <code>else</code>   | <code>goto</code>  | <code>short</code>    | <code>union</code>    |
| <code>char</code>     | <code>entry</code>  | <code>if</code>    | <code>sizeof</code>   | <code>unsigned</code> |
| <code>continue</code> | <code>enum</code>   | <code>int</code>   | <code>static</code>   | <code>void</code>     |
| <code>default</code>  | <code>extern</code> | <code>long</code>  | <code>struct</code>   | <code>while</code>    |

The `entry` keyword is not currently implemented by any compiler but is reserved for future use. Some implementations also reserve the words `fortran` and `asm`.

## D. Constants

There are several kinds of constants. Some of the more important constants are integer, long, character, floating, and enumeration. Hardware characteristics that affect sizes are summarized in “F. Hardware Characteristics” under part “LEXICAL CONVENTIONS”.

## Integer Constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with `0` (digit zero), decimal otherwise. A sequence of digits preceded by `0x` or `0X` (digit zero) is taken to be a hexadecimal integer.



The hexadecimal digits include `a` or `A` through `f` or `F` with values 10 through 15. A decimal constant whose value exceeds the largest signed machine integer is taken to be **long**; an octal or hex constant which exceeds the largest unsigned machine integer is likewise taken to be **long**.

### Explicit Long Constants

A decimal, octal, or hexadecimal integer constant immediately followed by `l` (letter ell) or `L` is a long constant. As discussed below, on some machines integer and long values may be considered identical.

### Character Constants

A character constant is a character enclosed in single quotes, as in `'x'`. The value of a character constant is the numerical value of the character in the machine's character set.

Certain nongraphic characters, the single quote (`'`) and the backslash (`\`), may be represented according to the following table of escape sequences:

|                 |                  |                   |
|-----------------|------------------|-------------------|
| new-line        | NL (LF)          | <code>\n</code>   |
| horizontal tab  | HT               | <code>\t</code>   |
| vertical tab    | VT               | <code>\v</code>   |
| backspace       | BS               | <code>\b</code>   |
| carriage return | CR               | <code>\r</code>   |
| form feed       | FF               | <code>\f</code>   |
| backslash       | <code>\</code>   | <code>\\</code>   |
| single quote    | <code>'</code>   | <code>\'</code>   |
| bit pattern     | <code>ddd</code> | <code>\ddd</code> |

The escape `\ddd` consists of the backslash followed by 1, 2, or 3 octal digits which are taken to specify the value of the desired character. A special case of this construction is `\0` (not followed by a digit), which indicates the character NUL. If the character following a backslash is not one of those specified, the backslash is ignored.

### Floating Constants

A floating constant consists of an integer part, a decimal point, a fraction part, an `e` or `E`, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing. Either the decimal point or the `e` and the exponent (not both) may be missing. Every floating constant is taken to be double precision.

### Enumeration Constants

Names declared as enumerators (see "E. Structure, Union, and Enumeration Declarations" in part "DECLARATIONS") are constants of the corresponding enumeration type. They behave like `int` constants.

## E. Strings

A string is a sequence of characters surrounded by double quotes, as in `"..."`. A string has type "array of characters" and storage class static (see part "NAMES") and is initialized with the given characters. All strings, even when written identically, are distinct. The compiler places a null byte (`\0`) at the end of each string so that programs which scan the string can find its end. In a string, the double quote character (`"`) must be preceded by a `\`; in addition, the same escapes as described for character constants may be used. Finally, a `\` and the immediately following new-line are ignored.

## F. Hardware Characteristics

The following table summarizes certain hardware properties that vary from machine to machine.



TABLE 3.A

## HARDWARE CHARACTERISTICS

|              | DEC PDP-11<br>ASCII | DEC VAX-11<br>ASCII | HONEYWELL 6000<br>ASCII | IBM 370<br>EBCDIC | INTERDATA 8/32<br>ASCII | WECO 3B<br>ASCII   |
|--------------|---------------------|---------------------|-------------------------|-------------------|-------------------------|--------------------|
| char         | 8 bits              | 8 bits              | 9 bits                  | 8 bits            | 8 bits                  | 8 bits             |
| int          | 16                  | 32                  | 36                      | 32                | 32                      | 32                 |
| short        | 16                  | 16                  | 36                      | 16                | 16                      | 16                 |
| long         | 32                  | 32                  | 36                      | 32                | 32                      | 32                 |
| float        | 32                  | 32                  | 36                      | 32                | 32                      | 32                 |
| double       | 64                  | 64                  | 72                      | 64                | 64                      | 64                 |
| float range  | $\pm 10^{\pm 38}$   | $\pm 10^{\pm 38}$   | $\pm 10^{\pm 38}$       | $\pm 10^{\pm 76}$ | $\pm 10^{\pm 76}$       | $\pm 10^{\pm 38}$  |
| double range | $\pm 10^{\pm 38}$   | $\pm 10^{\pm 38}$   | $\pm 10^{\pm 38}$       | $\pm 10^{\pm 76}$ | $\pm 10^{\pm 76}$       | $\pm 10^{\pm 308}$ |

## SYNTAX NOTATION

Syntactic categories are indicated by *italic* type and literal words and characters in **bold** type. Alternative categories are listed on separate lines. An optional terminal or nonterminal symbol is indicated by the subscript "opt," so that

$\{ \textit{expression}_{opt} \}$

indicates an optional expression enclosed in braces. The syntax is summarized in part "SYNTAX SUMMARY".

## NAMES

The C language bases the interpretation of an identifier upon two attributes of the identifier—its *storage class* and its *type*. The storage class determines the location and lifetime of the storage associated with an identifier; the type determines the meaning of the values found in the identifier's storage.

There are four declarable storage classes:

- automatic
- static
- external
- register.

Automatic variables are local to each invocation of a block (see "B. Compound Statement or Block" in part "STATEMENTS") and are discarded upon exit from the block. Static variables are local to a block but retain their values upon reentry to a block even after control has left the block. External variables exist and retain their values throughout the execution of the entire program and may be used for communication between functions, even separately compiled functions. Register variables are (if possible) stored in the fast registers of the machine; like automatic variables they are local to each block and disappear on exit from the block.

The C language supports several fundamental types of objects. Objects declared as characters (**char**) are large enough to store any member of the implementation's character set. If a genuine character from that character set is stored in a character variable, its value is equivalent to the integer code for that character. Other quantities may be stored into character variables, but the implementation is machine dependent.



Up to three sizes of integer, declared **short int**, **int**, and **long int**, are available. Longer integers provide no less storage than shorter ones, but the implementation may make either short integers or long integers, or both, equivalent to plain integers. "Plain" integers have the natural size suggested by the host machine architecture. The other sizes are provided to meet special needs.

Each enumeration (see "E. Structure, Union, and Enumeration Declarations" in part "DECLARATIONS") is conceptually a separate type with its own set of named constants. The properties of an **enum** type are identical to those of **int** type.

Unsigned integers, declared **unsigned**, obey the laws of arithmetic modulo  $2^n$  where  $n$  is the number of bits in the representation. (On the PDP-11, unsigned long quantities are not supported.)

Single-precision floating point (**float**) and double precision floating point (**double**) may be synonymous in some implementations.

Because objects of the foregoing types can usefully be interpreted as numbers, they will be referred to as *arithmetic* types. Types **char**, **int** of all sizes, and **enum** will collectively be called *integral* types. The **float** and **double** types will collectively be called *floating* types.

The **void** type specifies an empty set of values. It is used as the type returned by functions that generate no value.

Besides the fundamental arithmetic types, there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

- *arrays* of objects of most types
- *functions* which return objects of a given type
- *pointers* to objects of a given type
- *structures* containing a sequence of objects of various types
- *unions* capable of containing any one of several objects of various types.

In general these methods of constructing objects can be applied recursively.

## OBJECTS AND LVALUES

An *object* is a manipulatable region of storage. An *lvalue* is an expression referring to an object. An obvious example of an lvalue expression is an identifier. There are operators which yield lvalues: for example, if **E** is an expression of pointer type, then **\*E** is an lvalue expression referring to the object to which **E** points. The name "lvalue" comes from the assignment expression **E1 = E2** in which the left operand **E1** must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

## CONVERSIONS

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This part explains the result to be expected from such conversions. The conversions demanded by most ordinary operators are summarized under "F. Arithmetic Conversions". The summary will be supplemented as required by the discussion of each operator.



### A. Characters and Integers

A character or a short integer may be used wherever an integer may be used. In all cases the value is converted to an integer. Conversion of a shorter integer to a longer always involves sign extension; integers are signed quantities. Whether or not sign-extension occurs for characters is machine dependent, but it is guaranteed that a member of the standard character set is non-negative. Of the machines treated here, only the PDP-11 and VAX-11 sign-extend. On these machines, **char** variables range in value from -128 to 127. The more explicit type **unsigned char** forces the values to range from 0 to 255.

On machines that treat characters as signed, the characters of the ASCII set are all positive. However, a character constant specified with an octal escape suffers sign extension and may appear negative; for example, `'\377'` has the value -1.

When a longer integer is converted to a shorter or to a **char**, it is truncated on the left. Excess bits are simply discarded.

### B. Float and Double

All floating arithmetic in C is carried out in double precision. Whenever a **float** appears in an expression it is lengthened to **double** by zero padding its fraction. When a **double** must be converted to **float**, for example by an assignment, the **double** is rounded before truncation to **float** length.

### C. Floating and Integral

Conversions of floating values to integral type tend to be rather machine dependent. In particular the direction of truncation of negative numbers varies from machine to machine. The result is undefined if the value will not fit in the space provided.

Conversions of integral values to floating type are well behaved. Some loss of precision occurs if the destination lacks sufficient bits.

### D. Pointers and Integers

An expression of integral type may be added to or subtracted from a pointer; in such a case, the first is converted as specified in the discussion of the addition operator. Two pointers to objects of the same type may be subtracted; in this case, the result is converted to an integer as specified in the discussion of the subtraction operator.

### E. Unsigned

Whenever an unsigned integer and a plain integer are combined, the plain integer is converted to unsigned and the result is unsigned. The value is the least unsigned integer congruent to the signed integer (modulo  $2^{\text{wordsize}}$ ). In a 2's complement representation, this conversion is conceptual; and there is no actual change in the bit pattern.

When an unsigned integer is converted to **long**, the value of the result is the same numerically as that of the unsigned integer. Thus the conversion amounts to padding with zeros on the left.

### F. Arithmetic Conversions

A great many operators cause conversions and yield result types in a similar way. This pattern will be called the "usual arithmetic conversions."

- First, any operands of type **char** or **short** are converted to **int**, and any of type **float** are converted to **double**.



- Then, if either operand is **double**, the other is converted to **double** and that is the type of the result.
- Otherwise, if either operand is **long**, the other is converted to **long** and that is the type of the result.
- Otherwise, if either operand is **unsigned**, the other is converted to **unsigned** and that is the type of the result.
- Otherwise, both operands must be **int**, and that is the type of the result.

#### G. Void

The (nonexistent) value of a **void** object may not be used in any way, and neither explicit nor implicit conversion may be applied. Because a void expression denotes a nonexistent value, such an expression may be used only as an expression statement (see "A. Expression Statement" in part "STATEMENTS") or as the left operand of a comma expression (see "Comma Operator" in part "EXPRESSIONS").

An expression may be converted to type **void** by use of a cast. For example, this makes explicit the discarding of the value of a function call used as an expression statement.

### EXPRESSIONS

The precedence of expression operators is the same as the order of the major subsections of this section, highest precedence first. Thus, for example, the expressions referred to as the operands of + (see "D. Additive Operators") are those expressions defined under "A. Primary Expressions", "B. Unary Operators", and "C. Multiplicative Operators". Within each subpart, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators is summarized in the grammar of part "SYNTAX SUMMARY".

Otherwise, the order of evaluation of expressions is undefined. In particular the compiler considers itself free to compute subexpressions in the order it believes most efficient even if the subexpressions involve side effects. The order in which side effects take place is unspecified. Expressions involving a commutative and associative operator (\*, +, &, |, ^) may be rearranged arbitrarily even in the presence of parentheses; to force a particular order of evaluation, an explicit temporary must be used.

The handling of overflow and divide check in expression evaluation is machine dependent. Most existing implementations of C ignore integer overflows; treatment of division by 0 and all floating-point exceptions varies between machines and is usually adjustable by a library function.

#### A. Primary Expressions

Primary expressions involving ., ->, subscripting, and function calls group left to right.

```
primary-expression:
    identifier
    constant
    string
    ( expression )
    primary-expression [ expression ]
    primary-expression ( expression-listopt )
    primary-expression . identifier
    primary-expression -> identifier
```

```
expression-list:
    expression
    expression-list , expression
```



An identifier is a primary expression provided it has been suitably declared as discussed below. Its type is specified by its declaration. If the type of the identifier is "array of ...", however, then the value of the identifier expression is a pointer to the first object in the array; and the type of the expression is "pointer to ...". Moreover, an array identifier is not an lvalue expression. Likewise, an identifier which is declared "function returning ...", when used except in the function-name position of a call, is converted to "pointer to function returning ...".

A constant is a primary expression. Its type may be **int**, **long**, or **double** depending on its form. Character constants have type **int** and floating constants are **double**.

A string is a primary expression. Its type is originally "array of **char**", but following the same rule given above for identifiers, this is modified to "pointer to **char**" and the result is a pointer to the first character in the string. (There is an exception in certain initializers; see "F. Initialization" in part "DECLARATIONS".)

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

A primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. Usually, the primary expression has type "pointer to ...", the subscript expression is **int**, and the type of the result is "...". The expression  $E1[E2]$  is identical (by definition) to  $*((E1)+(E2))$ . All the clues needed to understand this notation are contained in this subpart together with the discussions in "B. Unary Operators" and "D. Additive Operators" on identifiers, **\*** and **+**, respectively. The implications are summarized under "C. Arrays, Pointers, and Subscripting" in part "TYPES REVISITED".

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The primary expression must be of type "function returning ...", and the result of the function call is of type "...". As indicated below, a hitherto unseen identifier followed immediately by a left parenthesis is contextually declared to represent a function returning an integer; thus in the most common case, integer-valued functions need not be declared.

Any actual arguments of type **float** are converted to **double** before the call. Any of type **char** or **short** are converted to **int**. Array names are converted to pointers. No other conversions are performed automatically; in particular, the compiler does not compare the types of actual arguments with those of formal arguments. If conversion is needed, use a cast; see "B. Unary Operators" and "G. Type Names" in part "DECLARATIONS".

In preparing for the call to a function, a copy is made of each actual parameter. Thus, all argument passing in C is strictly by value. A function may change the values of its formal parameters, but these changes cannot affect the values of the actual parameters. It is possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points. An array name is a pointer expression. The order of evaluation of arguments is undefined by the language; take note that the various compilers differ. Recursive calls to any function are permitted.

A primary expression followed by a dot followed by an identifier is an expression. The first expression must be a structure or a union, and the identifier must name a member of the structure or union. The value is the named member of the structure or union, and it is an lvalue if the first expression is an lvalue.

A primary expression followed by an arrow (built from **-** and **>**) followed by an identifier is an expression. The first expression must be a pointer to a structure or a union and the identifier must name a member of that structure or union. The result is an lvalue referring to the named member of the structure or union to which the pointer expression points. Thus the expression  $E1 \rightarrow MOS$  is the same as  $(*E1).MOS$ . Structures and unions are discussed in "E. Structure, Union, and Enumeration Declarations" in part "DECLARATIONS".

## B. Unary Operators

Expressions with unary operators group right to left.



```

unary-expression:
    * expression
    & lvalue
    - expression
    ! expression
    ~ expression
    ++ lvalue
    -- lvalue
    lvalue ++
    lvalue --
    ( type-name ) expression
    sizeof expression
    sizeof ( type-name )

```

The unary `*` operator means *indirection*: the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is "pointer to ...", the type of the result is "...".

The result of the unary `&` operator is a pointer to the object referred to by the lvalue. If the type of the lvalue is "...", the type of the result is "pointer to ...".

The result of the unary `-` operator is the negative of its operand. The usual arithmetic conversions are performed. The negative of an unsigned quantity is computed by subtracting its value from  $2^n$  where  $n$  is the number of bits in an `int`. There is no unary `+` operator.

The result of the logical negation operator `!` is 1 if the value of its operand is zero, zero if the value of its operand is nonzero. The type of the result is `int`. It is applicable to any arithmetic type or to pointers.

The `~` operator yields the 1's complement of its operand. The usual arithmetic conversions are performed. The type of the operand must be integral.

The object referred to by the lvalue operand of prefix `++` is incremented. The value is the new value of the operand but is not an lvalue. The expression `++x` is equivalent to `x+=1`. See the discussions "D. Additive Operators" and "N. Assignment Operators" for information on conversions.

The lvalue operand of prefix `--` is decremented analogously to the prefix `++` operator.

When postfix `++` is applied to an lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is incremented in the same manner as for the prefix `++` operator. The type of the result is the same as the type of the lvalue expression.

When postfix `--` is applied to an lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is decremented in the manner as for the prefix `--` operator. The type of the result is the same as the type of the lvalue expression.

An expression preceded by the parenthesized name of a data type causes conversion of the value of the expression to the named type. This construction is called a *cast*. Type names are described under "G. Type Names" in part "Declarations".

The `sizeof` operator yields the size in bytes of its operand. (A *byte* is undefined by the language except in terms of the value of `sizeof`. However, in all existing implementations, a byte is the space required to hold a `char`.) When applied to an array, the result is the total number of bytes in the array. The size is determined from the declarations of the objects in the expression. This expression is semantically an *unsigned constant*



and may be used anywhere a constant is required. Its major use is in communication with routines like storage allocators and I/O systems.

The `sizeof` operator may also be applied to a parenthesized type name. In that case it yields the size in bytes of an object of the indicated type.

The construction `sizeof(type)` is taken to be a unit, so the expression `sizeof(type)-2` is the same as `(sizeof(type))-2`.

### C. Multiplicative Operators

The multiplicative operators `*`, `/`, and `%` group left to right. The usual arithmetic conversions are performed.

*multiplicative expression:*

*expression \* expression*

*expression / expression*

*expression % expression*

The binary `*` operator indicates multiplication. The `*` operator is associative and expressions with several multiplications at the same level may be rearranged by the compiler.

The binary `/` operator indicates division. When positive integers are divided, truncation is toward 0; but the form of truncation is machine-dependent if either operand is negative. On all machines covered by this manual, the remainder has the same sign as the dividend. It is always true that  $(a/b)*b + a\%b$  is equal to  $a$  (if  $b$  is not 0).

The binary `%` operator yields the remainder from the division of the first expression by the second. The usual arithmetic conversions are performed. The operands must not be floating.

### D. Additive Operators

The additive operators `+` and `-` group left to right. The usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

*additive-expression:*

*expression + expression*

*expression - expression*

The result of the `+` operator is the sum of the operands. A pointer to an object in an array and a value of any integral type may be added. The latter is in all cases converted to an address offset by multiplying it by the length of the object to which the pointer points. The result is a pointer of the same type as the original pointer, and which points to another object in the same array, appropriately offset from the original object. Thus if  $P$  is a pointer to an object in an array, the expression  $P+1$  is a pointer to the next object in the array. No further type combinations are allowed for pointers.

The `+` operator is associative and expressions with several additions at the same level may be rearranged by the compiler.

The result of the `-` operator is the difference of the operands. The usual arithmetic conversions are performed. Additionally, a value of any integral type may be subtracted from a pointer, and then the same conversions for addition apply.

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an `int` representing the number of objects separating the pointed-to objects. This conversion



will in general give unexpected results unless the pointers point to objects in the same array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object length.

### E. Shift Operators

The shift operators `<<` and `>>` group left to right. Both perform the usual arithmetic conversions on their operands, each of which must be integral. Then the right operand is converted to `int`; the type of the result is that of the left operand. The result is undefined if the right operand is negative or greater than or equal to the length of the object in bits.

*shift-expression:*

```
expression << expression
expression >> expression
```

The value of `E1<<E2` is `E1` (interpreted as a bit pattern) left-shifted `E2` bits. Vacated bits are 0 filled. The value of `E1>>E2` is `E1` right-shifted `E2` bit positions. The right shift is guaranteed to be logical (0 fill) if `E1` is `unsigned`; otherwise, it may be arithmetic (fill by a copy of the sign bit).

### F. Relational Operators

The relational operators group left to right. This fact is not very useful; `a<b<c` does not mean what it seems to.

*relational-expression:*

```
expression < expression
expression > expression
expression <= expression
expression >= expression
```

The operators `<` (less than), `>` (greater than), `<=` (less than or equal to), and `>=` (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is `int`. The usual arithmetic conversions are performed. Two pointers may be compared; the result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same array.

### G. Equality Operators

*equality-expression:*

```
expression == expression
expression != expression
```

The `==` (equal to) and the `!=` (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus `a<b == c<d` is 1 whenever `a<b` and `c<d` have the same truth value.)

A pointer may be compared to an integer only if the integer is the constant 0. A pointer to which 0 has been assigned is guaranteed not to point to any object and will appear to be equal to 0. In conventional usage, such a pointer is considered to be null.

### H. Bitwise AND Operator

*and-expression:*

```
expression & expression
```

The `&` operator is associative, and expressions involving `&` may be rearranged. The usual arithmetic conversions are performed. The result is the bitwise AND function of the operands. The operator applies only to integral operands.



**I. Bitwise Exclusive OR Operator**

*exclusive-or-expression:*  
*expression ^ expression*

The ^ operator is associative, and expressions involving ^ may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral operands.

**J. Bitwise Inclusive OR Operator**

*inclusive-or-expression:*  
*expression | expression*

The | operator is associative, and expressions involving | may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands. The operator applies only to integral operands.

**K. Logical AND Operator**

*logical-and-expression:*  
*expression && expression*

The && operator groups left to right. It returns 1 if both its operands are nonzero, 0 otherwise. Unlike &, && guarantees left to right evaluation; moreover, the second operand is not evaluated if the first operand is 0.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always `int`.

**L. Logical OR Operator**

*logical-or-expression:*  
*expression || expression*

The || operator groups left to right. It returns 1 if either of its operands is nonzero, 0 otherwise. Unlike |, || guarantees left to right evaluation; moreover, the second operand is not evaluated if the value of the first operand is nonzero.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always `int`.

**M. Conditional Operator**

*conditional-expression:*  
*expression ? expression : expression*

Conditional expressions group right to left. The first expression is evaluated; and if it is nonzero, the result is the value of the second expression, otherwise that of third expression. If possible, the usual arithmetic conversions are performed to bring the second and third expressions to a common type. If both pointers are of the same type, the result has the common type. Otherwise, one must be a pointer and the other the constant 0, and the result has the type of the pointer. Only one of the second and third expressions is evaluated.



**N. Assignment Operators**

There are a number of assignment operators, all of which group right to left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place. The two parts of a compound assignment operator are separate tokens.

*assignment-expression:*

```

lvalue = expression
lvalue += expression
lvalue -= expression
lvalue *= expression
lvalue /= expression
lvalue %= expression
lvalue >>= expression
lvalue <<= expression
lvalue &= expression
lvalue ^= expression
lvalue != expression

```

In the simple assignment with `=`, the value of the expression replaces that of the object referred to by the lvalue. If both operands have arithmetic type, the right operand is converted to the type of the left preparatory to the assignment. Second, both operands may be structures or unions of the same type. Finally, if the left operand is a pointer, the right operand must in general be a pointer of the same type. However, the constant 0 may be assigned to a pointer; and it is guaranteed that this value will produce a null pointer distinguishable from a pointer to any object.

The behavior of an expression of the form `E1 op = E2` may be inferred by taking it as equivalent to `E1 = E1 op (E2)`; however, `E1` is evaluated only once. In `+=` and `-=`, the left operand may be a pointer; in which case, the (integral) right operand is converted as explained in "D. Additive Operators". All right operands and all nonpointer left operands must have arithmetic type.

**O. Comma Operator**

*comma-expression:*

```

expression , expression

```

A pair of expressions separated by a comma is evaluated left to right, and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left to right. In contexts where comma is given a special meaning, e.g., in lists of actual arguments to functions ("A. Primary Expressions") and lists of initializers ("F. Initialization" in part "Declarations"), the comma operator as described in this subpart can only appear in parentheses. For example,

```
f(a, (t=3, t+2), c)
```

has three arguments, the second of which has the value 5.

**DECLARATIONS**

Declarations are used to specify the interpretation which C gives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form

*declaration:*

```

decl-specifiers declarator-listopt ;

```



The declarators in the declarator-list contain the identifiers being declared. The decl-specifiers consist of a sequence of type and storage class specifiers.

*decl-specifiers:*

*type-specifier decl-specifiers<sub>opt</sub>*  
*sc-specifier decl-specifiers<sub>opt</sub>*

The list must be self-consistent in a way described below.

#### A. Storage Class Specifiers

The sc-specifiers are:

*sc-specifier:*

**auto**  
**static**  
**extern**  
**register**  
**typedef**

The **typedef** specifier does not reserve storage and is called a "storage class specifier" only for syntactic convenience. See "H. Typedef" for more information. The meanings of the various storage classes were discussed in part "Names".

The **auto**, **static**, and **register** declarations also serve as definitions in that they cause an appropriate amount of storage to be reserved. In the **extern** case, there must be an external definition (see part "External Definitions") for the given identifiers somewhere outside the function in which they are declared.

A **register** declaration is best thought of as an **auto** declaration, together with a hint to the compiler that the variables declared will be heavily used. Only the first few such declarations are effective. Moreover, only variables of certain types will be stored in registers; on the PDP-11, they are **int** or pointer. One other restriction applies to register variables: the address-of operator **&** cannot be applied to them. Smaller, faster programs can be expected if register declarations are used appropriately, but future improvements in code generation may render them unnecessary.

At most one sc-specifier may be given in a declaration. If the sc-specifier is missing from a declaration, it is taken to be **auto** inside a function, **extern** outside. Exception: functions are never automatic.

#### B. Type Specifiers

The type-specifiers are

*type-specifier:*

**char**  
**short**  
**int**  
**long**  
**unsigned**  
**float**  
**double**  
**void**  
*struct-or-union-specifier*  
*typedef-name*  
*enum-specifier*



The words **long**, **short**, and **unsigned** may be thought of as adjectives. The following combinations are acceptable.

```
short int
long int
unsigned int
unsigned char
long float
```

The meaning of the last is the same as **double**. Otherwise, at most one type-specifier may be given in a declaration. If the type-specifier is missing from a declaration, it is taken to be **int**.

Specifiers for structures, unions, and enumerations are discussed in "E. Structure, Union, and Enumeration Declarations". Declarations with **typedef** names are discussed in "H. Typedef".

### C. Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators, each of which may have an initializer.

```
declarator-list:
    init-declarator
    init-declarator , declarator-list
```

```
init-declarator:
    declarator initializeropt
```

Initializers are discussed in "F. Initialization". The specifiers in the declaration indicate the type and storage class of the objects to which the declarators refer. Declarators have the syntax:

```
declarator:
    identifier
    ( declarator )
    * declarator
    declarator ( )
    declarator [ constant-expressionopt ]
```

The grouping is the same as in expressions.

### D. Meaning of Declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type and storage class.

Each declarator contains exactly one identifier; it is this identifier that is declared. If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

A declarator in parentheses is identical to the unadorned declarator, but the binding of complex declarators may be altered by parentheses. See the examples below.

Now imagine a declaration

```
T D1
```

where **T** is a type-specifier (like **int**, etc.) and **D1** is a declarator. Suppose this declaration makes the identifier have type "... **T**," where the "..." is empty if **D1** is just a plain identifier (so that the type of **x** in "**int x**" is just **int**). Then if **D1** has the form

```
*D
```



the type of the contained identifier is "... pointer to T."

If D1 has the form

D()

then the contained identifier has the type "... function returning T."

If D1 has the form

D[constant-expression]

or

D[]

then the contained identifier has type "... array of T." In the first case, the constant expression is an expression whose value is determinable at compile time and whose type is **int**. (Constant expressions are defined precisely in part "Constant Expressions".) When several "array of" specifications are adjacent, a multidimensional array is created; the constant expressions which specify the bounds of the arrays may be missing only for the first member of the sequence. This elision is useful when the array is external and the actual definition, which allocates storage, is given elsewhere. The first constant expression may also be omitted when the declarator is followed by initialization. In this case the size is calculated from the number of initial elements supplied.

An array may be constructed from one of the basic types, from a pointer, from a structure or union, or from another array (to generate a multidimensional array).

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return arrays or functions although they may return pointers to such things; there are no arrays of functions although there may be arrays of pointers to functions. Likewise, a structure or union may not contain a function; but it may contain a pointer to a function.

As an example, the declaration

```
int i, *ip, f(), *fip(), (*pfi)();
```

declares an integer **i**, a pointer **ip** to an integer, a function **f** returning an integer, a function **fip** returning a pointer to an integer, and a pointer **pfi** to a function which returns an integer. It is especially useful to compare the last two. The binding of **\*fip()** is **\*(fip())**. The declaration suggests, and the same construction in an expression requires, the calling of a function **fip**. Using indirection through the (pointer) result to yield an integer. In the declarator **(\*pfi)()**, the extra parentheses are necessary, as they are also in an expression, to indicate that indirection through a pointer to a function yields a function, which is then called; it returns an integer.

As another example,

```
float fa[17], *afp[17];
```

declares an array of **float** numbers and an array of pointers to **float** numbers. Finally,

```
static int x3d[3][5][7];
```

declares a static 3-dimensional array of integers, with rank **3×5×7**. In complete detail, **x3d** is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the



expressions  $\times 3d$ ,  $\times 3d[i]$ ,  $\times 3d[i][j]$ ,  $\times 3d[i][j][k]$  may reasonably appear in an expression. The first three have type "array," the last has type `int`.

#### E. Structure, Union, and Enumeration Declarations

A structure is an object consisting of a sequence of named members. Each member may have any type. A union is an object which may, at a given time, contain any one of several members. Structure and union specifiers have the same form.

```
struct-or-union-specifier:
    struct-or-union { struct-decl-list }
    struct-or-union identifier { struct-decl-list }
    struct-or-union identifier
```

```
struct-or-union:
    struct
    union
```

The `struct-decl-list` is a sequence of declarations for the members of the structure or union:

```
struct-decl-list:
    struct-declaration
    struct-declaration struct-decl-list
```

```
struct-declaration:
    type-specifier struct-declarator-list ;
```

```
struct-declarator-list:
    struct-declarator
    struct-declarator , struct-declarator-list
```

In the usual case, a `struct-declarator` is just a declarator for a member of a structure or union. A structure member may also consist of a specified number of bits. Such a member is also called a *field*; its length is set off from the field name by a colon.

```
struct-declarator:
    declarator
    declarator : constant-expression
    : constant-expression
```

Within a structure, the objects declared have addresses which increase as the declarations are read left to right. Each nonfield member of a structure begins on an addressing boundary appropriate to its type; therefore, there may be unnamed holes in a structure. Field members are packed into machine integers; they do not straddle words. A field which does not fit into the space remaining in a word is put into the next word. No field may be wider than a word.

Fields are assigned right to left on the PDP-11 and VAX-11, left to right on other machines.

A `struct-declarator` with no declarator, only a colon and a width, indicates an unnamed field useful for padding to conform to externally-imposed layouts. As a special case, an unnamed field with a width of 0 specifies alignment of the next field at a word boundary. The "next field" presumably is a field, not an ordinary structure member because in the latter case the alignment would have been automatic.

The language does not restrict the types of things that are declared as fields, but implementations are not required to support any but integer fields. Moreover, even `int` fields may be considered to be unsigned. On the



PDP-11, fields are not signed and have only integer values; on the VAX-11, fields declared with `int` are treated as containing a sign. For these reasons, it is strongly recommended that fields be declared as **unsigned**. In all implementations, there are no arrays of fields, and the address-of operator `&` may not be applied to them, so that there are no pointers to fields.

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most one of the members can be stored in a union at any time.

A structure or union specifier of the second form, that is, one of

```
struct identifier { struct-decl-list }
union identifier { struct-decl-list }
```

declares the identifier to be the *structure tag* (or union tag) of the structure specified by the list. A subsequent declaration may then use the third form of specifier, one of

```
struct identifier
union identifier
```

Structure tags allow definition of self-referential structures. Structure tags also permit the long part of the declaration to be given once and used several times. It is illegal to declare a structure or union which contains an instance of itself, but a structure or union may contain a pointer to an instance of itself.

The names of members and tags do not conflict with each other or with ordinary variables. A particular name may not be used twice in the same structure, but the same name may be used in several different structures in the same scope.

A simple example of a structure declaration is

```
struct tnode
{
    char tword [20] ;
    int count;
    struct tnode *left;
    struct tnode *right;
};
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has been given, the declaration

```
struct tnode s, *sp;
```

declares `s` to be a structure of the given sort and `sp` to be a pointer to a structure of the given sort. With these declarations, the expression

```
sp->count
```

refers to the `count` field of the structure to which `sp` points;

```
s.left
```

refers to the left subtree pointer of the structure `s`; and

```
s.right->tword[0]
```



refers to the first character of the `tword` member of the right subtree of `s`.

Enumerations are unique types with named constants. However, the current language treats enumeration variables and constants as being of `int` type.

*enum-specifier:*

```
enum { enum-list }
enum identifier { enum-list }
enum identifier
```

*enum-list:*

```
enumerator
enum-list , enumerator
```

*enumerator:*

```
identifier
identifier = constant-expression
```

The identifiers in an `enum-list` are declared as constants and may appear wherever constants are required. If no enumerators with `=` appear, then the values of the corresponding constants begin at 0 and increase by 1 as the declaration is read from left to right. An enumerator with `=` gives the associated identifier the value indicated; subsequent identifiers continue the progression from the assigned value.

The names of enumerators in the same scope must all be distinct from each other and from those of ordinary variables.

The role of the identifier in the `enum-specifier` is entirely analogous to that of the structure tag in a `struct-specifier`; it names a particular enumeration. For example,

```
enum color { chartreuse, burgundy, claret=20, winedark };
...
enum color *cp, col;
...
col = claret;
cp = &col;
...
if (*cp == burgundy) ...
```

makes `color` the enumeration-tag of a type describing various colors, and then declares `cp` as a pointer to an object of that type, and `col` as an object of that type. The possible values are drawn from the set {0,1,20,21}.

#### F. Initialization

A declarator may specify an initial value for the identifier being declared. The initializer is preceded by `=` and consists of an expression or a list of values nested in braces.

*initializer:*

```
= expression
= { initializer-list }
= { initializer-list , }
```

*initializer-list:*

```
expression
initializer-list , initializer-list
{ initializer-list }
```

All the expressions in an initializer for a static or external variable must be constant expressions, which are described in part "CONSTANT EXPRESSIONS", or expressions which reduce to the address of a previously



declared variable, possibly offset by a constant expression. Automatic or register variables may be initialized by arbitrary expressions involving constants and previously declared variables and functions.

Static and external variables which are not initialized are guaranteed to start off as 0. Automatic and register variables which are not initialized are guaranteed to start off as garbage.

When an initializer applies to a *scalar* (a pointer or an object of arithmetic type), it consists of a single expression, perhaps in braces. The initial value of the object is taken from the expression; the same conversions as for assignment are performed.

When the declared variable is an *aggregate* (a structure or array), the initializer consists of a brace-enclosed, comma-separated list of initializers for the members of the aggregate written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the aggregate. If there are fewer initializers in the list than there are members of the aggregate, then the aggregate is padded with 0's. It is not permitted to initialize unions or automatic aggregates.

Braces may be elided as follows. If the initializer begins with a left brace, then the succeeding comma-separated list of initializers initializes the members of the aggregate; it is erroneous for there to be more initializers than members. If, however, the initializer does not begin with a left brace, then only enough elements from the list are taken to account for the members of the aggregate; any remaining members are left to initialize the next member of the aggregate of which the current aggregate is a part.

A final abbreviation allows a **char** array to be initialized by a string. In this case successive characters of the string initialize the members of the array.

For example,

```
int x [ ] = { 1, 3, 5 };
```

declares and initializes **x** as a 1-dimensional array which has three members, since no size was specified and there are three initializers.

```
float y [4] [3] =
{
    { 1, 3, 5 },
    { 2, 4, 6 },
    { 3, 5, 7 },
};
```

is a completely-bracketed initialization: 1, 3, and 5 initialize the first row of the array **y**[0], namely **y**[0][0], **y**[0][1], and **y**[0][2]. Likewise, the next two lines initialize **y**[1] and **y**[2]. The initializer ends early and therefore **y**[3] is initialized with 0. Precisely, the same effect could have been achieved by

```
float y [4] [3] =
{
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for **y** begins with a left brace but that for **y**[0] does not; therefore, three elements from the list are used. Likewise, the next three are taken successively for **y**[1] and **y**[2]. Also,

```
float y [4] [3] =
{
    { 1 }, { 2 }, { 3 }, { 4 }
};
```



initializes the first column of *y* (regarded as a 2-dimensional array) and leaves the rest 0.

Finally,

```
char msg[ ] = "Syntax error on line %s\n";
```

shows a character array whose members are initialized with a string.

#### G. Type Names

In two contexts (to specify type conversions explicitly by means of a cast and as an argument of `sizeof`), it is desired to supply the name of a data type. This is accomplished using a "type name", which in essence is a declaration for an object of that type which omits the name of the object.

*type-name:*

*type-specifier abstract-declarator*

*abstract-declarator:*

*empty*

*( abstract-declarator )*

*\* abstract-declarator*

*abstract-declarator ( )*

*abstract-declarator [ constant-expression<sub>opt</sub> ]*

To avoid ambiguity, in the construction

*( abstract-declarator )*

the abstract-declarator is required to be nonempty. Under this restriction, it is possible to identify uniquely the location in the abstract-declarator where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. For example,

```
int
int *
int * [3]
int (*) [3]
int *()
int *()()
```

name respectively the types "integer", "pointer to integer", "array of 3 pointers to integers", "pointer to an array of 3 integers", "function returning pointer to integer", and "pointer to function returning an integer".

#### H. Typedef

Declarations whose "storage class" is `typedef` do not define storage but instead define identifiers which can be used later as if they were type keywords naming fundamental or derived types.

*typedef-name:*

*identifier*

Within the scope of a declaration involving `typedef`, each identifier appearing as part of any declarator therein becomes syntactically equivalent to the type keyword naming the type associated with the identifier in the way described in "D. Meaning of Declarators". For example, after

```
typedef int MILES, *KLICKSP;
typedef struct { double re, im; } complex;
```



the constructions

```

MILES distance;
extern KLICKSP metricp;
complex z, *zp;

```

are all legal declarations; the type of **distance** is **int**, that of **metricp** is "pointer to **int**," and that of **z** is the specified structure. The **zp** is a pointer to such a structure.

The **typedef** does not introduce brand new types, only synonyms for types which could be specified in another way. Thus in the example above **distance** is considered to have exactly the same type as any other **int** object.

## STATEMENTS

Except as indicated, statements are executed in sequence.

### A. Expression Statement

Most statements are expression statements, which have the form

*expression ;*

Usually expression statements are assignments or function calls.

### B. Compound Statement or Block

So that several statements can be used where one is expected, the compound statement (also, and equivalently, called "block") is provided:

```

compound-statement:
    { declaration-listopt statement-listopt }

```

```

declaration-list:
    declaration
    declaration declaration-list

```

```

statement-list:
    statement
    statement statement-list

```

If any of the identifiers in the declaration-list were previously declared, the outer declaration is pushed down for the duration of the block, after which it resumes its force.

Any initializations of **auto** or **register** variables are performed each time the block is entered at the top. It is currently possible (but a bad practice) to transfer into a block; in that case the initializations are not performed. Initializations of **static** variables are performed only once when the program begins execution. Inside a block, **extern** declarations do not reserve storage so initialization is not permitted.

### C. Conditional Statement

The two forms of the conditional statement are

```

if ( expression ) statement
if ( expression ) statement else statement

```



In both cases, the expression is evaluated; and if it is nonzero, the first substatement is executed. In the second case, the second substatement is executed if the expression is 0. As usual the "else" ambiguity is resolved by connecting an **else** with the last encountered **else-less if**.

#### D. While Statement

The **while** statement has the form

```
while ( expression ) statement
```

The substatement is executed repeatedly so long as the value of the expression remains nonzero. The test takes place before each execution of the statement.

#### E. Do Statement

The **do** statement has the form

```
do statement while ( expression );
```

The substatement is executed repeatedly until the value of the expression becomes 0. The test takes place after each execution of the statement.

#### F. For Statement

The **for** statement has the form:

```
for ( expression-1opt ; expression-2opt ; expression-3opt ) statement
```

This statement is equivalent to

```
expression-1 ;  
while ( expression-2 )  
{  
    statement  
    expression-3 ;  
}
```

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0. The third expression often specifies an incrementing that is performed after each iteration.

Any or all of the expressions may be dropped. A missing *expression-2* makes the implied **while** clause equivalent to **while(1)**; other missing expressions are simply dropped from the expansion above.

#### G. Switch Statement

The **switch** statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form

```
switch ( expression ) statement
```

The usual arithmetic conversion is performed on the expression, but the result must be **int**. The statement is typically compound. Any statement within the statement may be labeled with one or more case prefixes as follows:

```
case constant-expression :
```



where the constant expression must be **int**. No two of the case constants in the same switch may have the same value. Constant expressions are precisely defined in part "CONSTANT EXPRESSIONS".

There may also be at most one statement prefix of the form

**default :**

When the **switch** statement is executed, its expression is evaluated and compared with each case constant. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case prefix. If no case constant matches the expression and if there is a **default**, prefix, control passes to the prefixed statement. If no case matches and if there is no **default**, then none of the statements in the switch is executed.

The prefixes **case** and **default** do not alter the flow of control, which continues unimpeded across such prefixes. To exit from a switch, see "H. Break Statement".

Usually, the statement that is the subject of a switch is compound. Declarations may appear at the head of this statement, but initializations of automatic or register variables are ineffective.

#### H. Break Statement

The statement

**break ;**

causes termination of the smallest enclosing **while**, **do**, **for**, or **switch** statement; control passes to the statement following the terminated statement.

#### I. Continue Statement

The statement

**continue ;**

causes control to pass to the loop-continuation portion of the smallest enclosing **while**, **do**, or **for** statement; that is to the end of the loop. More precisely, in each of the statements

|                    |                |                  |
|--------------------|----------------|------------------|
| <b>while (...)</b> | <b>do</b>      | <b>for (...)</b> |
| {                  | {              | {                |
| ...                | ...            | ...              |
| contin: ;          | contin: ;      | contin: ;        |
| }                  | } while (...); | }                |

a **continue** is equivalent to **goto contin**. (Following the **contin:** is a null statement, see "M. Null Statement".)

#### J. Return Statement

A function returns to its caller by means of the **return** statement, which has one of the forms

**return ;**  
**return expression ;**

In the first case, the returned value is undefined. In the second case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of the function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value.



**K. Goto Statement**

Control may be transferred unconditionally by means of the statement

*goto identifier;*

The identifier must be a label (see "L. Labeled Statement") located in the current function.

**L. Labeled Statement**

Any statement may be preceded by label prefixes of the form

*identifier:*

which serve to declare the identifier as a label. The only use of a label is as a target of a **goto**. The scope of a label is the current function, excluding any subblocks in which the same identifier has been redeclared. See part "SCOPE RULES".

**M. Null Statement**

The null statement has the form

;

A null statement is useful to carry a label just before the } of a compound statement or to supply a null body to a looping statement such as **while**.

**EXTERNAL DEFINITIONS**

A C program consists of a sequence of external definitions. An external definition declares an identifier to have storage class **extern** (by default) or perhaps **static**, and a specified type. The type-specifier (see "B. Type Specifiers" in part "DECLARATIONS") may also be empty, in which case the type is taken to be **int**. The scope of external definitions persists to the end of the file in which they are declared just as the effect of declarations persists to the end of a block. The syntax of external definitions is the same as that of all declarations except that only at this level may the code for functions be given.

**A. External Function Definitions**

Function definitions have the form

*function-definition:*

*decl-specifiers<sub>opt</sub> function-declarator function-body*

The only sc-specifiers allowed among the decl-specifiers are **extern** or **static**; see "B. Scope of Externals" in part "SCOPE RULES" for the distinction between them. A function declarator is similar to a declarator for a "function returning ..." except that it lists the formal parameters of the function being defined.

*function-declarator:*

*declarator ( parameter-list<sub>opt</sub> )*

*parameter-list:*

*identifier*

*identifier , parameter-list*

The function-body has the form

*function-body:*

*declaration-list compound-statement*



The identifiers in the parameter list, and only those identifiers, may be declared in the declaration list. Any identifiers whose type is not given are taken to be **int**. The only storage class which may be specified is **register**; if it is specified, the corresponding actual parameter will be copied, if possible, into a register at the outset of the function.

A simple example of a complete function definition is

```
int max(a, b, c)
    int a, b, c;
{
    int m;
    m = (a > b) ? a : b;
    return((m > c) ? m : c);
}
```

Here **int** is the type-specifier; **max(a, b, c)** is the function-declarator; **int a, b, c;** is the declaration-list for the formal parameters; { ... } is the block giving the code for the statement.

The C program converts all **float** actual parameters to **double**, so formal parameters declared **float** have their declaration adjusted to read **double**. Also, since a reference to an array in any context (in particular as an actual parameter) is taken to mean a pointer to the first element of the array, declarations of formal parameters declared "array of ..." are adjusted to read "pointer to ...".

#### B. External Data Definitions

An external data definition has the form

```
data-definition:
    declaration
```

The storage class of such data may be **extern** (which is the default) or **static** but not **auto** or **register**.

### SCOPE RULES

A C program need not all be compiled at the same time. The source text of the program may be kept in several files, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through explicit calls and through manipulation of external data.

Therefore, there are two kinds of scope to consider: first, what may be called the lexical scope of an identifier, which is essentially the region of a program during which it may be used without drawing "undefined identifier" diagnostics; and second, the scope associated with external identifiers, which is characterized by the rule that references to the same external identifier are references to the same object.

#### A. Lexical Scope

The lexical scope of identifiers declared in external definitions persists from the definition through the end of the source file in which they appear. The lexical scope of identifiers which are formal parameters persists through the function with which they are associated. The lexical scope of identifiers declared at the head of a block persists until the end of the block. The lexical scope of labels is the whole of the function in which they appear.

In all cases, however, if an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of that identifier outside the block is suspended until the end of the block.



Remember also ("E. Structure, Union, and Enumeration Declarations" in part "DECLARATIONS") that identifiers associated with ordinary variables on the one hand and those associated with structure and union members and tags on the other form two disjoint classes which do not conflict. Members and tags follow the same scope rules as other identifiers. The **typedef** names are in the same class as ordinary identifiers. They may be redeclared in inner blocks, but an explicit type must be given in the inner declaration:

```
typedef float distance;
```

```
...
```

```
auto int distance;
```

```
...
```

The **int** must be present in the second declaration, or it would be taken to be a declaration with no declarators and type **distance**.

## B. Scope of Externals

If a function refers to an identifier declared to be **extern**, then somewhere among the files or libraries constituting the complete program there must be an external definition for the identifier. All functions in a given program which refer to the same external identifier refer to the same object, so care must be taken that the type and size specified in the definition are compatible with those specified by each function which references the data.

The appearance of the **extern** keyword in an external definition indicates that storage for the identifiers being declared will be allocated in another file. Thus in a multifile program, an external data definition without the **extern** specifier must appear in exactly one of the files. Any other files which wish to give an external definition for the identifier must include the **extern** in the definition. The identifier can be initialized only in the declaration where storage is allocated.

Identifiers declared **static** at the top level in external definitions are not visible in other files. Functions may be declared **static**.

## COMPILER CONTROL LINES

The C compiler contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files. Lines beginning with **#** communicate with this preprocessor. These lines have syntax independent of the rest of the language; they may appear anywhere and have effect which lasts (independent of scope) until the end of the source program file.

### A. Token Replacement

A compiler-control line of the form

```
#define identifier token-string
```

causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens. Semicolons in or at the end of the token-string are part of that string. A line of the form

```
#define identifier( identifier , ... , identifier ) token-string
```

where there is no space between the first identifier and the **(**, is a macro definition with arguments. Subsequent instances of the first identifier followed by a **(**, a sequence of tokens delimited by commas, and a **)** are replaced



by the token string in the definition. Each occurrence of an identifier mentioned in the formal parameter list of the definition is replaced by the corresponding token string from the call. The actual arguments in the call are token strings separated by commas; however, commas in quoted strings or protected by parentheses do not separate arguments. The number of formal and actual parameters must be the same. Strings and character constants in the token-string are scanned for formal parameters, but strings and character constants in the rest of the program are not scanned for defined identifiers to replacement.

In both forms the replacement string is rescanned for more defined identifiers. In both forms a long definition may be continued on another line by writing \ at the end of the line to be continued.

This facility is most valuable for definition of "manifest constants," as in

```
#define TABSIZE 100
int table [TABSIZE];
```

A control line of the form

```
#undef identifier
```

causes the identifier's preprocessor definition to be forgotten.

#### B. File Inclusion

A compiler control line of the form

```
#include "filename"
```

causes the replacement of that line by the entire contents of the file *filename*. The named file is searched for first in the directory of the original source file and then in a sequence of specified or standard places. Alternatively, a control line of the form

```
#include <filename>
```

searches only the specified or standard places and not the directory of the source file. (How the places are specified is not part of the language.)

*#include's* may be nested.

#### C. Conditional Compilation

A compiler control line of the form

```
#if constant-expression
```

checks whether the constant expression evaluates to nonzero. (Constant expressions are discussed in part "CONSTANT EXPRESSIONS"; the following additional restriction applies here: the constant expression may not contain *sizeof* or an enumeration constant.) A control line of the form

```
#ifdef identifier
```

checks whether the identifier is currently defined in the preprocessor; i.e., whether it has been the subject of a *#define* control line. A control line of the form

```
#ifndef identifier
```



checks whether the identifier is currently undefined in the preprocessor.

All three forms are followed by an arbitrary number of lines, possibly containing a control line

**#else**

and then by a control line

**#endif**

If the checked condition is true, then any lines between **#else** and **#endif** are ignored. If the checked condition is false, then any lines between the test and a **#else** or, lacking a **#else**, the **#endif** are ignored.

These constructions may be nested.

#### D. Line Control

For the benefit of other preprocessors which generate C programs, a line of the form

**#line constant "filename"**

causes the compiler to believe, for purposes of error diagnostics, that the line number of the next source line is given by the constant and the current input file is named by the identifier. If the identifier is absent, the remembered file name does not change.

#### IMPLICIT DECLARATIONS

It is not always necessary to specify both the storage class and the type of identifiers in a declaration. The storage class is supplied by the context in external definitions and in declarations of formal parameters and structure members. In a declaration inside a function, if a storage class but no type is given, the identifier is assumed to be **int**; if a type but no storage class is indicated, the identifier is assumed to be **auto**. An exception to the latter rule is made for functions because **auto** functions do not exist. If the type of an identifier is "function returning ...", it is implicitly declared to be **extern**.

In an expression, an identifier followed by ( and not already declared is contextually declared to be "function returning int".

#### TYPES REVISITED

This part summarizes the operations which can be performed on objects of certain types.

##### A. Structures and Unions

Structures and unions may be assigned, passed as arguments to functions, and returned by functions. Other plausible operators, such as equality comparison and structure casts, are not implemented.

In a reference to a structure or union member, the name on the right must specify a member of the aggregate named or pointed to by the expression on the left. In general, a member of a union may not be inspected unless the value of the union has been assigned using that same member. However, one special guarantee is made by the language in order to simplify the use of unions: if a union contains several structures that share a common



initial sequence and if the union currently contains one of these structures, it is permitted to inspect the common initial part of any of the contained structures. For example, the following is a legal fragment:

```

union
{
    struct
    {
        int      type;
    } n;
    struct
    {
        int      type;
        int      intnode;
    } ni;
    struct
    {
        int      type;
        float    floatnode;
    } nf;
} u;
...
u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...
if (u.n.type == FLOAT)
    ... sin(u.nf.floatnode) ...

```

## B. Functions

There are only two things that can be done with a function—call it or take its address. If the name of a function appears in an expression not in the function-name position of a call, a pointer to the function is generated. Thus, to pass one function to another, one might say

```

int f();
...
g(f);

```

Then the definition of *g* might read

```

g(funcp)
    int (*funcp)();
{
    ...
    (*funcp)();
    ...
}

```

Notice that *f* must be declared explicitly in the calling routine since its appearance in *g(f)* was not followed by (.

## C. Arrays, Pointers, and Subscripting

Every time an identifier of array type appears in an expression, it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not lvalues. By definition, the subscript operator *[]* is interpreted in such a way that *E1[E2]* is identical to *\*((E1)+(E2))*. Because of the conversion rules which apply to *+*, if *E1* is an array and *E2* an integer, then *E1[E2]* refers to the *E2*-th member of *E1*. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.



A consistent rule is followed in the case of multidimensional arrays. If  $E$  is an  $n$ -dimensional array of rank  $i \times j \times \dots \times k$ , then  $E$  appearing in an expression is converted to a pointer to an  $(n-1)$ -dimensional array with rank  $j \times \dots \times k$ . If the  $*$  operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to  $(n-1)$ -dimensional array, which itself is immediately converted into a pointer.

If the  $*$  operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to  $(n-1)$ -dimensional array, which itself is immediately converted into a pointer.

For example, consider

```
int x[3][5];
```

Here  $x$  is a  $3 \times 5$  array of integers. When  $x$  appears in an expression, it is converted to a pointer to (the first of three) 5-membered arrays of integers. In the expression  $x[i]$ , which is equivalent to  $*(x+i)$ ,  $x$  is first converted to a pointer as described; then  $i$  is converted to the type of  $x$ , which involves multiplying  $i$  by the length of the object to which the pointer points, namely 5-integer objects. The results are added and indirection applied to yield an array (of five integers) which in turn is converted to a pointer to the first of the integers. If there is another subscript, the same argument applies again; this time the result is an integer.

It follows from all this that arrays in C are stored row-wise (last subscript varies fastest) and that the first subscript in the declaration helps determine the amount of storage consumed by an array but plays no other part in subscript calculations.

#### D. Explicit Pointer Conversions

Certain conversions involving pointers are permitted but have implementation-dependent aspects. They are all specified by means of an explicit type-conversion operator, see "B. Unary Operators" in part "EXPRESSIONS" and "G. Type Names" in part "DECLARATIONS".

A pointer may be converted to any of the integral types large enough to hold it. Whether an `int` or `long` is required is machine dependent. The mapping function is also machine dependent but is intended to be unsurprising to those who know the addressing structure of the machine. Details for some particular machines are given below.

An object of integral type may be explicitly converted to a pointer. The mapping always carries an integer converted from a pointer back to the same pointer but is otherwise machine dependent.

A pointer to one type may be converted to a pointer to another type. The resulting pointer may cause addressing exceptions upon use if the subject pointer does not refer to an object suitably aligned in storage. It is guaranteed that a pointer to an object of a given size may be converted to a pointer to an object of a smaller size and back again without change.

For example, a storage-allocation routine might accept a size (in bytes) of an object to allocate, and return a `char` pointer; it might be used in this way.

```
extern char *alloc();
double *dp;

dp = (double *) alloc(sizeof(double));
*dp = 22.0 / 7.0;
```

The `alloc` must ensure (in a machine-dependent way) that its return value is suitable for conversion to a pointer to `double`; then the use of the function is portable.

The pointer representation on the PDP-11 corresponds to a 16-bit integer and measures bytes. The `chars` have no alignment requirements; everything else must have an even address.



On the VAX-11, pointers are 32 bits long and measure bytes. Elementary objects are aligned on a boundary equal to their length, except that **double** quantities need be aligned only on even 4-byte boundaries. Aggregates are aligned on the strictest boundary required by any of their constituents.

On the Honeywell 6000, a pointer corresponds to a 36-bit integer; the word part is in the left 18 bits, and the two bits that select the character in a word lie just to their right. Thus **char** pointers measure units of  $2^{16}$  bytes; everything else is measured in units of  $2^{18}$  machine words. The **double** quantities and aggregates containing them must lie on an even word address ( $0 \bmod 2^{19}$ ).

The IBM 370 and the Interdata 8/32 are similar. On each, pointers are 32-bit quantities that measure bytes; elementary objects are aligned on a boundary equal to their length, so pointers to **short** must be  $0 \bmod 2$ , to **int** and **float**  $0 \bmod 4$ , and to **double**  $0 \bmod 8$ . Aggregates are aligned on the strictest boundary required by any of their constituents.

The 3B20 and 3B5 Processors characteristics are the same. On each, pointers are 24-bit quantities. Most objects are aligned on 4-byte boundaries. **Shorts** are aligned in all cases on 2-byte boundaries. Arrays of characters, all structures, **inits**, **longs**, **floats**, and **doubles** are aligned on 4-byte boundaries; but structure members may be packed tighter.

## CONSTANT EXPRESSIONS

In several places C requires expressions which evaluate to a constant: after **case**, as array bounds, and in initializers. In the first two cases, the expression can involve only integer constants, character constants, enumeration constants, and **sizeof** expressions, possibly connected by the binary operators

+ - \* / % & ! ^ << >> == != < > <= >=

or by the unary operators

-

or by the ternary operator

?:

Parentheses can be used for grouping but not for function calls.

More latitude is permitted for initializers; besides constant expressions as discussed above, one can also apply the unary & operator to external or static objects and to external or static arrays subscripted with a constant expression. The unary & can also be applied implicitly by appearance of unsubscripted arrays and functions. The basic rule is that initializers must evaluate either to a constant or to the address of a previously declared external or static object plus or minus a constant.

Less latitude is allowed for constant expressions after **#if**; **sizeof** expressions and enumeration constants are not permitted.

## PORTABILITY CONSIDERATIONS

Certain parts of C are inherently machine dependent. The following list of potential trouble spots is not meant to be all-inclusive but to point out the main ones.

Purely hardware issues like word size and the properties of floating point arithmetic and integer division have proven in practice to be not much of a problem. Other facets of the hardware are reflected in differing



implementations. Some of these, particularly sign extension (converting a negative character into a negative integer) and the order in which bytes are placed in a word, are a nuisance that must be carefully watched. Most of the others are only minor problems.

The number of **register** variables that can actually be placed in registers varies from machine to machine as does the set of valid types. Nonetheless, the compilers all do things properly for their own machine; excess or invalid **register** declarations are ignored.

Some difficulties arise only when dubious coding practices are used. It is exceedingly unwise to write programs that depend on any of these properties.

The order of evaluation of function arguments is not specified by the language. It is right to left on the PDP-11 and VAX-11; left to right on the others. The order in which side effects take place is also unspecified.

Since character constants are really objects of type **int**, multicharacter character constants may be permitted. The specific implementation is very machine dependent because the order in which characters are assigned to a word varies from one machine to another.

Fields are assigned to words and characters to integers right to left on the PDP-11 and VAX-11 and left to right on other machines. These differences are invisible to isolated programs which do not indulge in type running (e.g., by converting an **int** pointer to a **char** pointer and inspecting the pointed-to storage) but must be accounted for when conforming to externally-imposed storage layouts.

The language accepted by the various compilers differs in minor details. Most notably, the current PDP-11 compiler will not initialize structures containing bit fields and does not accept a few assignment operators in certain contexts where the value of the assignment is used.

#### ANACHRONISMS

Because C is an evolving language, certain obsolete constructions may be found in older programs. Although some versions of the compiler support such anachronisms, they have by and large disappeared leaving only a portability problem behind.

Earlier versions of C used the form `=op` instead of `op=` for assignment operators. This leads to ambiguities, typified by

```
x=-1
```

which assigns -1 to **x**, but previously decremented **x**.

The syntax of initializers has changed. Previously, the equals sign that introduces an initializer was not present, so instead of

```
int    x = 1;
```

one used

```
int    x 1;
```

The change was made because the initialization

```
int    f(1)
```

resembles a function declaration closely enough to confuse the compilers.



A structure or union member reference is a chain of member references (qualifications) that are prefixed by either a pointer to a structure or union or a structure or union proper. Because each qualification implies the addition of an offset within an address computation, older compilers (which failed to check for membership in the appropriate structure or union) allowed omission of those qualifications with an offset of zero. Complete qualification is now required.

Previous versions of the compiler were lax in detecting mixed assignments involving pointers and arithmetic quantities. These are now remarked upon.

## SYNTAX SUMMARY

This summary of C syntax is intended more for aiding comprehension than as an exact statement of the language.

### A. Expressions

The basic expressions are:

*expression:*

```

primary
* expression
& lvalue
- expression
! expression
~ expression
++ lvalue
-- lvalue
lvalue ++
lvalue --
sizeof expression
( type-name ) expression
expression binop expression
expression ? expression : expression
lvalue asgnop expression
expression , expression

```

*primary:*

```

identifier
constant
string
( expression )
primary ( expression-listopt )
primary [ expression ]
primary . identifier
primary -> identifier

```

*lvalue:*

```

identifier
primary [ expression ]
lvalue . identifier
primary -> identifier
* expression
( lvalue )

```

The primary-expression operators

( ) [ ] . ->



have highest priority and group left to right. The unary operators

\* & - ! ~ ++ -- sizeof ( type-name )

have priority below the primary operators but higher than any binary operator and group right to left. Binary operators group left to right; they have priority decreasing as indicated below. The conditional operator groups right to left.

*binop:*

```

*      /      %
+      -
>>    <<
<      >      <=      >=
==     !=
&
^
|
&&
||
?:

```

Assignment operators all have the same priority and all group right to left.

*asgnop:*

```

=      +=      -=      *=      /=      %=      >>=      <<=      &=      ^=      |=

```

The comma operator has the lowest priority and groups left to right.

## B. Declarations

*declaration:*

*decl-specifiers* *init-declarator-list*<sub>opt</sub> ;

*decl-specifiers:*

*type-specifier* *decl-specifiers*<sub>opt</sub>  
*sc-specifier* *decl-specifiers*<sub>opt</sub>

*sc-specifier:*

auto  
static  
extern  
register  
typedef

*type-specifier:*

char  
short  
int  
long  
unsigned  
float  
double  
void  
*struct-or-union-specifier*

*typedef-name*

*enum-specifier*



*enum-specifier:*

**enum** { enum-list }  
**enum** identifier { enum-list }  
**enum** identifier

*enum-list:*

enumerator  
enum-list , enumerator

*enumerator:*

identifier  
identifier = constant-expression

*init-declarator-list:*

init-declarator  
init-declarator , init-declarator-list

*init-declarator:*

declarator initializer<sub>opt</sub>

*declarator:*

identifier  
( declarator )  
\* declarator  
declarator ( )  
declarator [ constant-expression<sub>opt</sub> ]

*struct-or-union-specifier:*

**struct** { struct-decl-list }  
**struct** identifier { struct-decl-list }  
**struct** identifier  
**union** { struct-decl-list }  
**union** identifier { struct-decl-list }  
**union** identifier

*struct-decl-list:*

struct-declaration  
struct-declaration struct-decl-list

*struct-declaration:*

type-specifier struct-declarator-list ;

*struct-declarator-list:*

struct-declarator  
struct-declarator , struct-declarator-list

*struct-declarator:*

declarator  
declarator : constant-expression  
: constant-expression



*initializer:*

*= expression*  
*= { initializer-list }*  
*= { initializer-list , }*

*initializer-list:*

*expression*  
*initializer-list , initializer-list*  
*{ initializer-list }*

*type-name:*

*type-specifier abstract-declarator*

*abstract-declarator:*

*empty*  
*( abstract-declarator )*  
*\* abstract-declarator*  
*abstract-declarator ( )*  
*abstract-declarator [ constant-expression<sub>opt</sub> ]*

*typedef-name:*

*identifier*

### C. Statements

*compound-statement:*

*{ declaration-list<sub>opt</sub> statement-list<sub>opt</sub> }*

*declaration-list:*

*declaration*  
*declaration declaration-list*

*statement-list:*

*statement*  
*statement statement-list*

*statement:*

*compound-statement*  
*expression ;*  
*if ( expression ) statement*  
*if ( expression ) statement    else statement*  
*while ( expression ) statement*  
*do statement    while ( expression ) ;*  
*for ( expression-1<sub>opt</sub> ; expression-2<sub>opt</sub> ; expression-3<sub>opt</sub> ) statement*  
*switch ( expression ) statement*  
*case constant-expression :    statement*  
*default : statement*  
*break ;*  
*continue ;*  
*return ;*  
*return expression ;*  
*goto identifier ;*  
*identifier : statement*  
*;*



## D. External Definitions

*program:*  
    *external-definition*  
    *external-definition program*

*external-definition:*  
    *function-definition*  
    *data-definition*

*function-definition:*  
    *type-specifier*<sub>opt</sub> *function-declarator* *function-body*

*function-declarator:*  
    *declarator* ( *parameter-list*<sub>opt</sub> )

*parameter-list:*  
    *identifier*  
    *identifier* , *parameter-list*

*function-body:*  
    *declaration-list* *compound-statement*

*data-definition:*  
    **extern**<sub>opt</sub> *type-specifier*<sub>opt</sub> *init-declarator-list*<sub>opt</sub> ;  
    **static**<sub>opt</sub> *type-specifier*<sub>opt</sub> *init-declarator-list*<sub>opt</sub> ;

## E. Preprocessor

**#define** *identifier* *token-string* .  
**#define** *identifier* ( *identifier* , ... , *identifier* ) *token-string*  
**#undef** *identifier*  
**#include** " *filename* "  
**#include** < *filename* >  
**#if** *constant-expression*  
**#ifdef** *identifier*  
**#ifndef** *identifier*  
**#else**  
**#endif**  
**#line** *constant* " *filename* "



## LIBRARIES

## A. General

This part describes the libraries that are supported on the UNIX operating system. A library is a collection of related functions and/or declarations that simplify programming effort. All of the functions described are also described in Part 3 of the UNIX System User's Manual. Most of the declarations described are in Part 3 of the UNIX System User's Manual. The three main libraries on the UNIX system are:

|                     |                                                                                                                                                                                                                                               |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>C library</b>    | This is the basic library for C language programs. The C library is composed of functions and declarations used for file access, string testing and manipulation, character testing and manipulation, memory allocation, and other functions. |
| <b>Object file</b>  | This library provides functions for the access and manipulation of object files.                                                                                                                                                              |
| <b>Math library</b> | This library provides exponential, bessel functions, logarithmic, hyperbolic, and trigonometric functions.                                                                                                                                    |

Some libraries consist of two portions—functions and declarations. In some cases, the user must request that the functions (and/or declarations) of a specific library be included in a program being compiled. In other cases, the functions (and/or declarations) are included automatically.

## Including Functions

When a program is being compiled, the compiler will automatically search the C language library to locate and include functions that are used in the program. This is the case only for the C library and no other library. In order for the compiler to locate and include functions from other libraries, the user must specify these libraries on the command line for the compiler. For example, when using functions of the math library, the user must request that the math library be searched by including the argument `-lm` on the command line, such as:

```
cc file.c -lm
```

This method should be used for all functions that are not part of the C language library.

## Including Declarations

Some functions require a set of declarations in order to operate properly. The declarations of all libraries must be included by request of the user. A set of declarations is stored in a file under the `/usr/include` directory. These files are referred to as *header files*. In order to include a certain header file, the user must specify this request within the C language program. The request is in the form:

```
#include <file.h>
```

where *file* is the name of the file. Since this request is handled by the preprocessor, header files should appear at the beginning of the (first) file being compiled.

The remainder of this part describes the functions and header files of the various libraries. The description of each library begins with the actions required by the user to include the functions and/or header files in a program being compiled (if any). Following the description of the actions required, is information in three column format of the form:

|                 |                     |                           |
|-----------------|---------------------|---------------------------|
| <b>function</b> | <b>reference(N)</b> | <b>Brief description.</b> |
|-----------------|---------------------|---------------------------|

The functions are grouped by type while the reference refers to section 'N' in the UNIX System User's Manual. Following this, are descriptions of the header files associated with these functions (if any).



## B. The C Library

The C library consists of several types of functions. All the functions of the C library are loaded automatically by the compiler. Various declarations must be included by the user as required. The functions of the C library are divided into the following types:

- input/output control
- string manipulation
- character manipulation
- time functions
- miscellaneous functions.

### Input/Output Control

These functions of the C library are included as needed during the compiling of a C language program automatically. No command line request is needed.

The header file required by the input/output functions should be included in the program being compiled. This is accomplished by including the line:

```
#include <stdio.h>
```

near the beginning of the (first) file being compiled.

The input/output functions are grouped into the following categories:

- file access
- file status
- input
- output
- miscellaneous.

### File Access Functions

| <i>FUNCTION</i> | <i>REFERENCE</i>  | <i>BRIEF DESCRIPTION</i>                                                                         |
|-----------------|-------------------|--------------------------------------------------------------------------------------------------|
| <b>fclose</b>   | <b>fclose(3S)</b> | Close an open stream.                                                                            |
| <b>fdopen</b>   | <b>fopen(3S)</b>  | Associate stream with an <b>open(2)</b> ed file.                                                 |
| <b>fileno</b>   | <b>ferror(3S)</b> | Integer associated with an open stream.                                                          |
| <b>fopen</b>    | <b>fopen(3S)</b>  | Open a stream with specified permissions. A "stream" is defined to be what <b>fopen</b> returns. |
| <b>freopen</b>  | <b>fopen(3S)</b>  | Substitute named file in place of open stream.                                                   |



|               |                   |                                                              |
|---------------|-------------------|--------------------------------------------------------------|
| <b>fseek</b>  | <b>fseek(3S)</b>  | Reposition stream pointer.                                   |
| <b>pclose</b> | <b>popen(3S)</b>  | Close a stream opened by <b>popen</b> .                      |
| <b>popen</b>  | <b>popen(3S)</b>  | Create pipe as a stream between calling process and command. |
| <b>rewind</b> | <b>fseek(3S)</b>  | Reposition stream pointer at beginning of file.              |
| <b>setbuf</b> | <b>setbuf(3S)</b> | Assign buffering to stream.                                  |

*File Status Functions*

| <b>FUNCTION</b> | <b>REFERENCE</b>  | <b>BRIEF DESCRIPTION</b>            |
|-----------------|-------------------|-------------------------------------|
| <b>clearerr</b> | <b>ferror(3S)</b> | Reset error condition on stream.    |
| <b>feof</b>     | <b>ferror(3S)</b> | Test for "end of file" on stream.   |
| <b>ferror</b>   | <b>ferror(3S)</b> | Test for error condition on stream. |
| <b>ftell</b>    | <b>fseek(3S)</b>  | Return current stream pointer.      |

*Input Functions*

| <b>FUNCTION</b> | <b>REFERENCE</b>  | <b>BRIEF DESCRIPTION</b>                  |
|-----------------|-------------------|-------------------------------------------|
| <b>fgetc</b>    | <b>getc(3S)</b>   | True function for <b>getc</b> (3S).       |
| <b>fgets</b>    | <b>gets(3S)</b>   | Read string from stream.                  |
| <b>fread</b>    | <b>fread(3S)</b>  | General buffered read from stream.        |
| <b>fscanf</b>   | <b>scanf(3S)</b>  | Read using format from stream.            |
| <b>getc</b>     | <b>getc(3S)</b>   | Return next character from stream.        |
| <b>getchar</b>  | <b>getc(3S)</b>   | Return next character from <b>stdin</b> . |
| <b>gets</b>     | <b>gets(3S)</b>   | Read string from <b>stdin</b> .           |
| <b>getw</b>     | <b>getc(3S)</b>   | Read word from stream.                    |
| <b>scanf</b>    | <b>scanf(3S)</b>  | Read using format from <b>stdin</b> .     |
| <b>sscanf</b>   | <b>scanf(3S)</b>  | Read using format from string.            |
| <b>ungetc</b>   | <b>ungetc(3S)</b> | Put back one character on stream.         |

*Output Functions*

| <b>FUNCTION</b> | <b>REFERENCE</b>  | <b>BRIEF DESCRIPTION</b>                             |
|-----------------|-------------------|------------------------------------------------------|
| <b>fflush</b>   | <b>fclose(3S)</b> | Write all currently buffered characters from stream. |



|                      |                         |                                               |
|----------------------|-------------------------|-----------------------------------------------|
| <code>fprintf</code> | <code>printf(3S)</code> | Print using format to stream.                 |
| <code>fputc</code>   | <code>putc(3S)</code>   | True function for <code>putc</code> (3S).     |
| <code>fputs</code>   | <code>puts(3S)</code>   | Write string to stream.                       |
| <code>fwrite</code>  | <code>fread(3S)</code>  | General buffered write to stream.             |
| <code>printf</code>  | <code>printf(3S)</code> | Print using format to <code>stdout</code> .   |
| <code>putc</code>    | <code>putc(3S)</code>   | Write next character to stream.               |
| <code>putchar</code> | <code>putc(3S)</code>   | Write next character to <code>stdout</code> . |
| <code>puts</code>    | <code>puts(3S)</code>   | Write string to <code>stdout</code> .         |
| <code>putw</code>    | <code>putc(3S)</code>   | Write word to stream.                         |
| <code>sprintf</code> | <code>printf(3S)</code> | Write using format to string.                 |

### Miscellaneous Functions

| FUNCTION             | REFERENCE                | BRIEF DESCRIPTION                                      |
|----------------------|--------------------------|--------------------------------------------------------|
| <code>ctermid</code> | <code>ctermid(3S)</code> | Return file name for controlling terminal.             |
| <code>cuserid</code> | <code>cuserid(3S)</code> | Return login name for owner of current process.        |
| <code>system</code>  | <code>system(3S)</code>  | Execute system command.                                |
| <code>tempnam</code> | <code>tmpnam(3S)</code>  | Create temporary file name using directory and prefix. |
| <code>tmpnam</code>  | <code>tmpnam(3S)</code>  | Create temporary file name.                            |
| <code>tmpfile</code> | <code>tmpfile(3S)</code> | Create temporary file.                                 |

### Input/Output Header File (`stdio.h`)

The following listing is the contents of `stdio.h` for the 3B20S Processor. Note that along with parameters used by the `stdio` functions several macros are defined. The following files are open automatically by the system:

```

stdin    standard input file
stdout   standard output file
stderr   standard error file

```

Also, note that the functions `fopen`, `fdopen`, and `freopen` are declared as returning a structure of type `FILE`; `fgets` and `gets` are declared as returning character pointers; and `ftell` is declared as returning a long integer.

```

/* this example is included as */
/* a typical illustration of the */
/* header file stdio.h */
#ifdef _NFILE
#define _NFILE 20

```



```

#define BUFSIZ          512
typedef struct {
    unsigned char *_ptr;
    int _cnt;
    unsigned char *_base;
    char _flag;
    char _file;
} FILE;
#define _IOREAD 01
#define _IOWRT 02
#define _IONBF 04
#define _IOMYBUF 010
#define _IOEOF 020
#define _IOERR 040
#define _IOUNK 0100 /* indicates buffering status unknown */
#define _IORW 0200
#ifndef NULL
#define NULL 0
#endif
#ifndef EOF
#define EOF (-1)
#endif
#define stdin (&_iob[0])
#define stdout (&_iob[1])
#define stderr (&_iob[2])
#ifndef lint
#define getc(p) (((p)->_cnt) >= 0)?((int) *((p)->_ptr++): _filbuf(p))
#endif
#define getchar() getc(stdin)
#ifndef lint
#define putc(x,p) (((p)->_cnt) >= 0)?((int)) (*((p)->_ptr)++ = (unsigned char)(x)): \
    _flsbuf((unsigned char)(x),p))
#endif
#define putchar(x) putc(x,stdout)
#define feof(p) (((p)->_flag & _IOEOF) != 0)
#define ferror(p) (((p)->_flag & _IOERR) != 0)
#define fileno(p) p->file
extern FILE _iob[_NFILE];
extern FILE *fopen( );
extern FILE *fdopen( );
extern FILE *freopen( );
extern long ftell( );
extern char *fgets( );
extern char *gets( );
#define L_ctermid 9
#define L_cuserid 9
#define P_tmpdir "/usr/tmp/"
#define L_tmpnam 10+sizeof(P_tmpdir)
#endif

```

#### String Manipulation Functions

These functions are used to locate characters within a string, copy, concatenate, and compare strings. These functions are located and loaded during the compiling of a C language program automatically. No command line



request is needed since these functions are part of the C library. There are no declarations associated with these functions.

| <i>FUNCTION</i> | <i>REFERENCE</i>  | <i>BRIEF DESCRIPTION</i>                                         |
|-----------------|-------------------|------------------------------------------------------------------|
| <b>strcat</b>   | <b>string(3C)</b> | Concatenate two strings.                                         |
| <b>strchr</b>   | <b>string(3C)</b> | Search string for character.                                     |
| <b>strcmp</b>   | <b>string(3C)</b> | Compares two strings.                                            |
| <b>strcpy</b>   | <b>string(3C)</b> | Copy string over string.                                         |
| <b>strcspn</b>  | <b>string(3C)</b> | Length of initial string not containing set of characters.       |
| <b>strlen</b>   | <b>string(3C)</b> | Length of string.                                                |
| <b>strncat</b>  | <b>string(3C)</b> | Concatenate two strings with fixed length.                       |
| <b>strncmp</b>  | <b>string(3C)</b> | Compares two strings with fixed length.                          |
| <b>strncpy</b>  | <b>string(3C)</b> | Copy string over string with fixed length.                       |
| <b>strpbrk</b>  | <b>string(3C)</b> | Search string for any set of characters.                         |
| <b>strrchr</b>  | <b>string(3C)</b> | Search string backwards for character.                           |
| <b>strspn</b>   | <b>string(3C)</b> | Length of initial string containing set of characters.           |
| <b>strtok</b>   | <b>string(3C)</b> | Search string for token separated by any of a set of characters. |

#### Character Manipulation

The following functions and declarations are used for testing and translating ASCII characters. These functions are located and loaded automatically during the compiling of a C language program. No command line request is needed since these functions are part of the C library.

The declarations associated with these functions should be included in the program being compiled. This is accomplished by including the line:

```
#include <ctype.h>
```

near the beginning of the (first) file being compiled.

#### Character Testing Functions

These functions can be used to identify characters as uppercase or lowercase letters, digits, punctuation, etc.

| <i>FUNCTION</i> | <i>REFERENCE</i> | <i>BRIEF DESCRIPTION</i>   |
|-----------------|------------------|----------------------------|
| <b>isalnum</b>  | <b>ctype(3C)</b> | Is character alphanumeric? |
| <b>isalpha</b>  | <b>ctype(3C)</b> | Is character alphabetic?   |



|                 |                  |                                                    |
|-----------------|------------------|----------------------------------------------------|
| <b>isascii</b>  | <b>ctype(3C)</b> | Is integer ASCII character?                        |
| <b>iscntrl</b>  | <b>ctype(3C)</b> | Is character a control character?                  |
| <b>isdigit</b>  | <b>ctype(3C)</b> | Is character a digit?                              |
| <b>isgraph</b>  | <b>ctype(3C)</b> | Is character a printing character?                 |
| <b>islower</b>  | <b>ctype(3C)</b> | Is character a lowercase letter?                   |
| <b>isprint</b>  | <b>ctype(3C)</b> | Is character a printing character including space? |
| <b>ispunct</b>  | <b>ctype(3C)</b> | Is character a punctuation character?              |
| <b>isspace</b>  | <b>ctype(3C)</b> | Is character a white space character?              |
| <b>isupper</b>  | <b>ctype(3C)</b> | Is character an uppercase letter?                  |
| <b>isxdigit</b> | <b>ctype(3C)</b> | Is character a hex digit?                          |

### **Character Translation Functions**

These functions provide translation of uppercase to lowercase, lowercase to uppercase, and integer to ASCII.

| <b>FUNCTION</b> | <b>REFERENCE</b> | <b>BRIEF DESCRIPTION</b>            |
|-----------------|------------------|-------------------------------------|
| <b>toascii</b>  | <b>conv(3C)</b>  | Convert integer to ASCII character. |
| <b>tolower</b>  | <b>conv(3C)</b>  | Convert character to lowercase.     |
| <b>toupper</b>  | <b>conv(3C)</b>  | Convert character to uppercase.     |

### **Character Header File (ctype.h)**

The following listing is the *ctype.h* file which is located in the */usr/include* directory. This file is included in a program with the line:

```
#include <ctype.h>
```

This file provides a few data declarations and defines the macros.

```
/* this example is included as */
/* a typical illustration of the */
/* header file ctype.h */
#define _U 01
#define _L 02
#define _N 04
#define _S 010
#define _P 020
#define _C 040
#define _B 0100
#define _X 0200
extern char _ctype[];
```



```

#define isalpha(c)    ((_ctype+1)[c]&(_U|_L))
#define isupper(c)    ((_ctype+1)[c]&_U)
#define islower(c)    ((_ctype+1)[c]&_L)
#define isdigit(c)    ((_ctype+1)[c]&_N)
#define isxdigit(c)   ((_ctype+1)[c]&_X)
#define isspace(c)    ((_ctype+1)[c]&_S)
#define ispunct(c)    ((_ctype+1)[c]&_P)
#define isalnum(c)    ((_ctype+1)[c]&(_U|_L|_N))
#define isprint(c)    ((_ctype+1)[c]&(_P|_U|_L|_N|_B))
#define isgraph(c)    ((_ctype+1)[c]&(_P|_U|_L|_N))
#define iscntrl(c)    ((_ctype+1)[c]&_C)
#define isascii(c)    ((unsigned char)(c)<=0177)
#define _toupper(c)   ((c)-'a'+'A')
#define _tolower(c)   ((c)-'A'+'a')
#define toascii(c)    ((c)&0177)

```

### Time Functions

These functions are used for accessing and reformatting the systems idea of the current date and time. These functions are located and loaded automatically during the compiling of a C language program. No command line request is needed since these functions are part of the C library.

The header file associated with these functions should be included in the program being compiled. This is accomplished by including the line:

```
#include <time.h>
```

near the beginning of the (first) file being compiled.

These functions (except `tzset`) convert a time such as returned by `time(2)`.

| FUNCTION               | REFERENCE              | BRIEF DESCRIPTION                                                  |
|------------------------|------------------------|--------------------------------------------------------------------|
| <code>asctime</code>   | <code>ctime(3C)</code> | Return string representation of date and time.                     |
| <code>ctime</code>     | <code>ctime(3C)</code> | Return string representation of date and time, given integer form. |
| <code>gmtime</code>    | <code>ctime(3C)</code> | Return Greenwich Mean Time.                                        |
| <code>localtime</code> | <code>ctime(3C)</code> | Return local time.                                                 |
| <code>tzset</code>     | <code>ctime(3C)</code> | Set time zone field from environment variable.                     |

### Time Header File (*time.h*)

The following listing is the *time.h* file which is located in the */usr/include* directory. The *tm* structure is the type of structure returned by `gmtime` and `localtime`. Note that the `gmtime` and `localtime` functions are declared as returning pointers to structures of type *tm* and `ctime` and `asctime` are declared as returning character pointers.

The long *timezone* variable contains the difference (in seconds) between GMT and local standard time. For EST, this difference is 18,000 seconds. The *daylight* variable is nonzero if Daylight Savings Time conversion



should be applied. The *tzname* variable defines the time zone names. For EST, the declaration would be `char *tzname [2] = " EST", " EDT";`

If the environment variable *TZ* is present, *asctime* uses the contents of the variable to override the default time zone. The value of *TZ* must be a 3-letter time zone name, followed by a number representing the difference between GMT and local time (in hours). Following the time difference is an optional 3-letter name for a daylight time zone. For example, for users in EST, the value of *TZ* would be *EST5EDT*. By setting the *TZ* variable, the values of *timezone*, *daylight*, and *tzname* are changed accordingly.

```
/* this example is included as */
/* a typical illustration of the */
/* header file time.h */
struct tm {
int tm_sec;
int tm_min;
int tm_hour; /* hour of day (0 to 24) */
int tm_mday; /* day of month (1 to 31) */
int tm_mon; /* month of year (0 to 11) */
int tm_year; /* last two digits of current year */
int tm_wday; /* day of week (Sunday = 0) */
int tm_yday; /* day of year (0 to 365) */
int tm_isdst; /* nonzero if DST in effect */
};
extern struct tm *gmtime( ), *localtime( );
extern char *ctime( ), *asctime( );
extern void tzset( );
extern long timezone;
extern int daylight;
extern char *tzname[ ];
```

### Miscellaneous Functions

These functions support a wide variety of operations. Some of these are numerical conversion, password file and group file access, memory allocation, random number generation, and table management. These functions are located and included in a program being compiled automatically. No command line request is needed since these functions are part of the C library.

Some of these functions require declarations to be included. These are described following the descriptions of the functions.

### Numerical Conversion

The following functions perform numerical conversion.

| FUNCTION | REFERENCE | BRIEF DESCRIPTION                          |
|----------|-----------|--------------------------------------------|
| a64l     | a64l(3C)  | Convert string to base 64 ASCII.           |
| atof     | atof(3C)  | Convert string to floating.                |
| atoi     | atoi(3C)  | Convert string to integer.                 |
| atol     | atol(3C)  | Convert string to long.                    |
| frexp    | frexp(3C) | Split floating into mantissa and exponent. |



|              |                  |                                           |
|--------------|------------------|-------------------------------------------|
| <b>l3tol</b> | <b>l3tol(3C)</b> | Convert 3-byte integer to long.           |
| <b>lto13</b> | <b>l3tol(3C)</b> | Convert long to 3-byte integer.           |
| <b>ldexp</b> | <b>frexp(3C)</b> | Combine mantissa and exponent.            |
| <b>l64a</b>  | <b>a641(3C)</b>  | Convert base 64 ASCII to string.          |
| <b>modf</b>  | <b>frexp(3C)</b> | Split mantissa into integer and fraction. |

**DES Algorithm Access**

The following functions allow access to the DES algorithm used on the UNIX operating system. The DES algorithm is implemented with variations to frustrate use of hardware implementations of the DES for key search.

| <b>FUNCTION</b> | <b>REFERENCE</b> | <b>BRIEF DESCRIPTION</b>                          |
|-----------------|------------------|---------------------------------------------------|
| <b>crypt</b>    | <b>crypt(3C)</b> | Encode string using salt.                         |
| <b>encrypt</b>  | <b>crypt(3C)</b> | Encode/decode string of 0's and 1's.              |
| <b>setkey</b>   | <b>crypt(3C)</b> | Initialize for subsequent use of <b>encrypt</b> . |

**Group File Access**

The following functions are used to obtain entries from the group file. Declarations for these functions must be included in the program being compiled with the line:

```
#include <grp.h>
```

| <b>FUNCTION</b> | <b>REFERENCE</b>    | <b>BRIEF DESCRIPTION</b>              |
|-----------------|---------------------|---------------------------------------|
| <b>endgrent</b> | <b>getgrent(3C)</b> | Close group file being processed.     |
| <b>getgrent</b> | <b>getgrent(3C)</b> | Get next group file entry.            |
| <b>getgrgid</b> | <b>getgrent(3C)</b> | Return next group with matching gid.  |
| <b>getgrnam</b> | <b>getgrent(3C)</b> | Return next group with matching name. |
| <b>setgrent</b> | <b>getgrent(3C)</b> | Rewind group file being processed.    |

**Password File Access**

These functions are used to search and access information stored in the password file (/etc/passwd). Some functions require declarations that can be included in the program being compiled by adding the line:

```
#include <pwd.h>
```

| <b>FUNCTION</b> | <b>REFERENCE</b>    | <b>BRIEF DESCRIPTION</b>             |
|-----------------|---------------------|--------------------------------------|
| <b>endpwent</b> | <b>getpwent(3C)</b> | Close password file being processed. |
| <b>getpw</b>    | <b>getpw(3C)</b>    | Search password file for uid.        |



|                 |                     |                                       |
|-----------------|---------------------|---------------------------------------|
| <b>getpwent</b> | <b>getpwent(3C)</b> | Get next password file entry.         |
| <b>getpwnam</b> | <b>getpwent(3C)</b> | Return next entry with matching name. |
| <b>getpwuid</b> | <b>getpwent(3C)</b> | Return next entry with matching uid.  |
| <b>putpwent</b> | <b>putpwent(3C)</b> | Write entry on stream.                |
| <b>setpwent</b> | <b>getpwent(3C)</b> | Rewind password file being accessed.  |

### **Parameter Access**

The following functions provide access to several different types of parameters. None require any declarations.

| <b>FUNCTION</b> | <b>REFERENCE</b>   | <b>BRIEF DESCRIPTION</b>                                   |
|-----------------|--------------------|------------------------------------------------------------|
| <b>getopt</b>   | <b>getopt(3C)</b>  | Get next option from option list.                          |
| <b>getcwd</b>   | <b>getcwd(3C)</b>  | Return string representation of current working directory. |
| <b>getenv</b>   | <b>getenv(3C)</b>  | Return string value associated with environment variable.  |
| <b>getpass</b>  | <b>getpass(3C)</b> | Read string from terminal without echoing.                 |

### **Hash Table Management**

The following functions are used to manage hash search tables.

| <b>FUNCTION</b> | <b>REFERENCE</b>   | <b>BRIEF DESCRIPTION</b>     |
|-----------------|--------------------|------------------------------|
| <b>hcreate</b>  | <b>hsearch(3C)</b> | Create hash table.           |
| <b>hdestroy</b> | <b>hsearch(3C)</b> | Destroy hash table.          |
| <b>hsearch</b>  | <b>hsearch(3C)</b> | Search hash table for entry. |

### **Binary Tree Management**

The following functions are used to manage a binary tree.

| <b>FUNCTION</b> | <b>REFERENCE</b>   | <b>BRIEF DESCRIPTION</b>        |
|-----------------|--------------------|---------------------------------|
| <b>tdelete</b>  | <b>tsearch(3C)</b> | Deletes nodes from binary tree. |
| <b>tsearch</b>  | <b>tsearch(3C)</b> | Search binary tree.             |
| <b>twalk</b>    | <b>tsearch(3C)</b> | Walk binary tree.               |

### **Table Management**

The following functions are used to manage a table. The "table" is basically a 2-dimensional character array. The first subscript defines the maximum number of entries in the table. The second subscript defines the width



(or length) of a single entry. Since none of these functions allocate storage, sufficient memory must be allocated before using these functions.

| <i>FUNCTION</i> | <i>REFERENCE</i> | <i>BRIEF DESCRIPTION</i>                 |
|-----------------|------------------|------------------------------------------|
| bsearch         | bsearch(3C)      | Search table using binary search.        |
| lsearch         | lsearch(3C)      | Search table using linear search.        |
| qsort           | qsort(3C)        | Sort table using quicker-sort algorithm. |

### *Memory Allocation*

The following functions provide a means by which memory can be dynamically allocated or freed.

| <i>FUNCTION</i> | <i>REFERENCE</i> | <i>BRIEF DESCRIPTION</i>           |
|-----------------|------------------|------------------------------------|
| calloc          | malloc(3C)       | Allocate zeroed storage.           |
| free            | malloc(3C)       | Free previously allocated storage. |
| malloc          | malloc(3C)       | Allocate storage.                  |
| realloc         | malloc(3C)       | Change size of allocated storage.  |

### *Pseudorandom Number Generation*

The following functions are used to generate pseudorandom numbers. The functions that end with 48 are a family of interfaces to a pseudorandom number generator based upon the linear congruential algorithm and 48-bit integer arithmetic. The **rand** and **srand** functions provide an interface to a multiplicative congruential random number generator with period of 232.

| <i>FUNCTION</i> | <i>REFERENCE</i> | <i>BRIEF DESCRIPTION</i>                                              |
|-----------------|------------------|-----------------------------------------------------------------------|
| drand48         | drand48(3C)      | Random double over the interval [0 to 1).                             |
| lcong48         | drand48(3C)      | Set parameters for drand48, lrand48, and mrand48.                     |
| lrand48         | drand48(3C)      | Random long over the interval [0 to 2 <sup>31</sup> ).                |
| mrnd48          | drand48(3C)      | Random long over the interval [-2 <sup>31</sup> to 2 <sup>31</sup> ). |
| rand            | rand(3C)         | Random integer over the interval [0 to 214).                          |
| seed48          | drand48(3C)      | Seed the generator for drand48, lrand48, and mrand48.                 |
| srand           | rand(3C)         | Seed the generator for rand.                                          |
| srand48         | drand48(3C)      | Seed the generator for drand48, lrand48, and mrand48 using a long.    |



*Signal Handling Functions*

The functions `gsignal` and `ssignal` implement a software facility similar to `signal(2)` in the UNIX System User's Manual. This facility enables users to indicate the disposition of error conditions and allows users to handle signals for their own purposes. The declarations associated with these functions can be included in the program being compiled by the line

```
#include <signal.h>
```

These declarations define ASCII names for the 15 software signals.

| <i>FUNCTION</i>      | <i>REFERENCE</i>         | <i>BRIEF DESCRIPTION</i>                  |
|----------------------|--------------------------|-------------------------------------------|
| <code>gsignal</code> | <code>ssignal(3C)</code> | Send a software signal.                   |
| <code>ssignal</code> | <code>ssignal(3C)</code> | Arrange for handling of software signals. |

*Miscellaneous*

The following functions do not fall into any previously described category.

| <i>FUNCTION</i>      | <i>REFERENCE</i>         | <i>BRIEF DESCRIPTION</i>                                             |
|----------------------|--------------------------|----------------------------------------------------------------------|
| <code>abort</code>   | <code>abort(3C)</code>   | Cause an IOT signal to be sent to the process.                       |
| <code>abs</code>     | <code>abs(3C)</code>     | Return the absolute integer value.                                   |
| <code>ecvt</code>    | <code>ecvt(3C)</code>    | Convert double to string.                                            |
| <code>fcvt</code>    | <code>ecvt(3C)</code>    | Convert double to string using Fortran format.                       |
| <code>gcvt</code>    | <code>ecvt(3C)</code>    | Convert double to string using Fortran F or E format.                |
| <code>isatty</code>  | <code>ttynam(3C)</code>  | Test whether integer file descriptor is associated with a terminal.  |
| <code>mktemp</code>  | <code>mktemp(3C)</code>  | Create file using template.                                          |
| <code>monitor</code> | <code>monitor(3C)</code> | Cause process to record a histogram of program counter location.     |
| <code>swab</code>    | <code>swab(3C)</code>    | Swap and copy bytes.                                                 |
| <code>ttynam</code>  | <code>ttynam(3C)</code>  | Return pathname of terminal associated with integer file descriptor. |

*Group File Header File (grp.h)*

The `grp.h` file provides the structure used by several group file functions. This file can be included in a program by adding the line:

```
#include <grp.h>
```



```

/* this example is included as */
/* a typical illustration of the */
/* header file grp.h */
struct group {
char *gr_name; /* group name */
char *gr_passwd; /* group password */
int gr_gid; /* group id */
char **gr_mem; /* list of pointers to group members */
};

```

### **Password File Header File (*pwd.h*)**

The following listing describes the contents of *pwd.h* which is used by several of the password file functions. This file can be included in the program with the line:

```
#include <pwd.h>
```

The *pw\_comment* field is actually a structure of type *comment*.

```

/* this example is included as */
/* a typical illustration of the */
/* header file pwd.h */
struct passwd {
char *pw_name; /* login name */
char *pw_passwd; /* password */
int pw_uid; /* user id */
int pw_gid; /* group id */
char *pw_age; /* age of password */
char *pw_comment; /* comments */
char *pw_gecos; /* optional GCOS user id */
char *pw_dir; /* login directory */
char *pw_shell; /* shell used by this login */
};
struct comment {
char *c_dept; /* user's department */
char *c_name; /* user's name */
char *c_acct; /* user's account number */
char *c_bin; /* user's mail bin */
};

```

### **Signal Handling Header File (*signal.h*)**

The following listing describes the *signal.h* file which is under the */usr/include* directory. This file can be included by adding the line:

```
#include <signal.h>
```

The *signal.h* file contains the declarations used by the functions that handle signals. Most of this file defines *names* to each numerical signal.

```

/* this example is included as */
/* a typical illustration of the */

```



```

/* header file signal.h */
#define SIGHUP 1 /* hangup */
#define SIGINT 2 /* interrupt (rubout) */
#define SIGQUIT 3 /* quit (ASCII FS) */
#define SIGILL 4 /* illegal instruction (not reset when caught) */
#define SIGTRAP 5 /* trace trap (not reset when caught) */
#define SIGIOT 6 /* IOT instruction */
#define SIGEMT 7 /* EMT instruction */
#define SIGFPE 8 /* floating point exception */
#define SIGKILL 9 /* kill (cannot be caught or ignored) */
#define SIGBUS 10 /* bus error */
#define SIGSEGV 11 /* segmentation violation */
#define SIGSYS 12 /* bad argument to system call */
#define SIGPIPE 13 /* write on a pipe with no one to read it */
#define SIGALRM 14 /* alarm clock */
#define SIGTERM 15 /* software termination signal from kill */
#define SIGUSR1 16 /* user defined signal 1 */
#define SIGUSR2 17 /* user defined signal 2 */
#define SIGCLD 18 /* death of a child */
#define SIGPWR 19 /* power-fail restart */
#define NSIG 20
#define SIG_DFL (int (*)( ))0
#if lint
#define SIG_IGN (int (*)( ))0
#else
#define SIG_IGN (int (*)( ))1
#endif
extern (*signal( ))( );

```

### C. The Object File Library

The object file library provides functions to access object files. The functions allow access closing to single object files or object files that are part of an archive. Some functions locate portions of an object file such as the symbol table, the file header, sections, and lines within functions. Other functions read these types of entries into memory.

This library consists of several portions. The functions reside in */usr/lib/libld.a* and are located and loaded during the compiling of a C language program by a command line request. The form of this request is:

```
cc file -lld
```

which causes the link editor to search the object file library.

In addition, various header files must be included. This is accomplished by including the line:

```
#include <sgs/file>
```

where *file* is the name of the appropriate header file. The header files required for each function are defined at the beginning of each function description. Following the descriptions of the functions is a description of the header files. The **HEADER** macro returns a pointer to the **HEADER** field of the **LDFILE** structure pointed to by *ldptr*. The **HEADER** field is the file header structure of the object file. The **IOPTR** macro returns the contents of the **IOPTR** field of the **LDFILE** structure pointed to by *ldptr*. The **IOPTR** field contains a file pointer returned by **fopen** and used by the input/output functions of the C library. The **TYPE** macro returns the **TYPE**



field of the *LDFILE* structure pointed to by *ldptr*. The *TYPE* field contains the file magic number which is used to distinguish between archive members and simple object files.

| <i>FUNCTION</i>  | <i>REFERENCE</i>     | <i>BRIEF DESCRIPTION</i>                                                                                     |
|------------------|----------------------|--------------------------------------------------------------------------------------------------------------|
| <i>ldaclose</i>  | <i>ldclose(3X)</i>   | Close object file being processed.                                                                           |
| <i>ldahread</i>  | <i>ldahread(3X)</i>  | Read archive header.                                                                                         |
| <i>ldaopen</i>   | <i>ldopen(3X)</i>    | Open object file for reading.                                                                                |
| <i>ldclose</i>   | <i>ldclose(3X)</i>   | Close object file being processed.                                                                           |
| <i>ldfhread</i>  | <i>ldfhread(3X)</i>  | Read file header of object file being processed.                                                             |
| <i>ldlinit</i>   | <i>ldlread(3X)</i>   | Prepare object file for reading line number entries via <i>ldlitem</i> .                                     |
| <i>ldlitem</i>   | <i>ldlread(3X)</i>   | Read line number entry from object file after <i>ldlinit</i> .                                               |
| <i>ldlread</i>   | <i>ldlread(3X)</i>   | Read line number entry from object file.                                                                     |
| <i>ldlseek</i>   | <i>ldlseek(3X)</i>   | Seeks to the line number entries of the object file being processed.                                         |
| <i>ldnlseek</i>  | <i>ldlseek(3X)</i>   | Seeks to the line number entries of the object file being processed given name.                              |
| <i>ldnrseek</i>  | <i>ldrseek(3X)</i>   | Seeks to the relocation entries of the object file being processed given name.                               |
| <i>ldnshread</i> | <i>ldshread(3X)</i>  | Read section header of the named section of the object file being processed.                                 |
| <i>ldnsseek</i>  | <i>ldsseek(3X)</i>   | Seeks to the section of the object file being processed given name.                                          |
| <i>ldohseek</i>  | <i>ldohseek(3X)</i>  | Seeks to the optional file header of the object file being processed.                                        |
| <i>ldopen</i>    | <i>ldopen(3X)</i>    | Open object file for reading.                                                                                |
| <i>ldrseek</i>   | <i>ldrseek(3X)</i>   | Seeks to the relocation entries of the object file being processed.                                          |
| <i>ldshread</i>  | <i>ldshread(3X)</i>  | Read section header of an object file being processed.                                                       |
| <i>ldsseek</i>   | <i>ldsseek(3X)</i>   | Seeks to the section of the object file being processed.                                                     |
| <i>ldtbindex</i> | <i>ldtbindex(3X)</i> | Returns the long index of the symbol table entry at the current position of the object file being processed. |
| <i>ldtbread</i>  | <i>ldtbread(3X)</i>  | Reads the symbol table entry specified by <i>symindex</i> of the object file being processed.                |
| <i>ldtbseek</i>  | <i>ldtbseek(3X)</i>  | Seeks to the symbol table of the object file being processed.                                                |



#### D. The Math Library

The math library consists of functions and a header file. The functions are located and loaded during the compiling of a C language program by a command line request. The form of this request is:

```
cc file -lm
```

which causes the link editor to search the math library. In addition to the request to load the functions, the header file of the math library should be included in the program being compiled. This is accomplished by including the line:

```
#include <math.h>
```

near the beginning of the (first) file being compiled.

The functions are grouped into the following categories:

- trigonometric functions
- bessel functions
- hyperbolic functions
- miscellaneous functions.

#### *Trigonometric Functions*

These functions are used to compute angles (in decimal radian measure), sines, cosines, and tangents. All of these values are expressed in double precision and should always be declared as such.

| <i>FUNCTION</i> | <i>REFERENCE</i> | <i>BRIEF DESCRIPTION</i>                                         |
|-----------------|------------------|------------------------------------------------------------------|
| acos            | trig(3M)         | Return arc cosine.                                               |
| asin            | trig(3M)         | Return arc sine.                                                 |
| atan            | trig(3M)         | Return arc tangent.                                              |
| atan2           | trig(3M)         | Return arc tangent of a ratio.                                   |
| cos             | trig(3M)         | Return cosine.                                                   |
| hypot           | hypot(3M)        | Return the square root of the sum of the squares of two numbers. |
| sin             | trig(3M)         | Return sine.                                                     |
| tan             | trig(3M)         | Return tangent.                                                  |

#### *Bessel Functions*

These functions calculate bessel functions of the first and second kinds of several orders for real values. The bessel functions are j0, j1, jn, y0, y1, and yn. The functions are located in section `bessel(3M)`.



*Hyperbolic Functions*

These functions are used to compute the hyperbolic sine, cosine, and tangent for real values.

| <b>FUNCTION</b> | <b>REFERENCE</b> | <b>BRIEF DESCRIPTION</b>   |
|-----------------|------------------|----------------------------|
| cosh            | sinh(3M)         | Return hyperbolic cosine.  |
| sinh            | sinh(3M)         | Return hyperbolic sine.    |
| tanh            | sinh(3M)         | Return hyperbolic tangent. |

*Miscellaneous Functions*

These functions cover a wide variety of operations, such as natural logarithm, exponential, and absolute value. In addition, several are provided to truncate the decimal portion of double precision numbers.

| <b>FUNCTION</b> | <b>REFERENCE</b> | <b>BRIEF DESCRIPTION</b>                                                               |
|-----------------|------------------|----------------------------------------------------------------------------------------|
| ceil            | floor(3M)        | Returns the smallest integer not less than a given value.                              |
| exp             | exp(3M)          | Returns the exponential function of a given value.                                     |
| fabs            | floor(3M)        | Returns the absolute value of a given value.                                           |
| floor           | floor(3M)        | Returns the largest integer not greater than a given value.                            |
| fmod            | floor(3M)        | Returns the remainder produced by the division of two given values.                    |
| gamma           | gamma(3M)        | Returns the natural log of gamma as a function of the absolute value of a given value. |
| log             | exp(3M)          | Returns the natural logarithm of a given value.                                        |
| pow             | exp(3M)          | Returns the result of a given value raised to another given value.                     |
| sqrt            | exp(3M)          | Returns the square root of a given value.                                              |

*Math Header File (math.h)*

The following listing is the *math.h* file which is located in the */usr/include* directory. Note that all the math functions are declared as returning double-precision values. Also note that *HUGE* is defined as

1.701411733192644270 x 10<sup>38</sup>

```

/* this example is included as */
/* a typical illustration of the */
/* math header file (math.h) */
extern double fabs( ), floor( ), ceil( ), fmod( ), ldexp( );
extern double sqrt( ), hypot( ), atof( );
extern double sin( ), cos( ), tan( ), asin( ), acos( ), atan( ), atan2( );

```



```
extern double exp( ), log( ), log10( ), pow( );
extern double sinh( ), cosh( ), tanh( );
extern double gamma( );
extern double j0( ), j1( ), jn( ), y0( ), y1( ), yn( );
#define HUGE      1.701411733192644270e38
```

## THE "cc" COMMAND

### A. General

The C compiler `cc(1)` is used to compile C language programs or assembly language programs into machine language. This document briefly describes the usage of the C compiler. Most of this information is in the UNIX System User's Manual.

### B. Usage

The `cc` command is invoked as:

```
cc options files
```

where *options* control the compiling; *files* are the files to be compiled.

The following options are interpreted by `cc`. See `ld(1)` for link editor options.

- `-c` Suppresses the link edit phase of the compilation and forces an object file to be produced even if only one program is compiled.
- `-Dname=def` Defines the *name* to the preprocessor, as if by `#define`. If no definition is given, the name is defined as 1.
- `-Dname` Defines the *name* to the preprocessor, as if by `#define`. If no definition is given, the name is defined as 1.
- `-E` Runs only the macro preprocessor on the named C language programs and sends the result to the standard output.
- `-g` Arranges for the compiler to produce additional information needed for the use of `sdb(1)`.
- `-Idir` Changes the algorithm for searching for `#include` files whose names do not begin with `/` to look in *dir* before looking in the directories on the standard list. Thus, `#include` files whose names are enclosed in "`..`" will be searched for first in the directory of the *file* argument, then in directories named in the `-I` options, and last in directories on a standard list. For `#include` files whose names are enclosed by `<...>`, the directory of the *file* argument is not searched.
- `-O` Invokes an object-code optimizer. The optimizer will move, merge, and delete code so symbolic debugging with line numbers could be confusing when the optimizer is used.
- `-p` Arranges for the compiler to produce code which counts the number of times each routine is called. Also, if link editing takes place, replaces the standard startoff routine by one which automatically calls `monitor(3C)` at the start and arranges to write out a `mon.out` file at normal termination of execution of the object program. An execution profile can be generated by use of `prof(1)`.
- `-P` Runs only the macro preprocessor on the named C language programs and leaves the result on corresponding files suffixed `.i`.



- S Compiles the named C language programs and leaves the assembler-language output on corresponding files suffixed .s.
- U *name* Removes any initial definition of *name*, where *name* is a reserved symbol that is predefined by the particular preprocessor. The current list of these possibly reserved symbols includes the operating system [unix (this reserved symbol refers to the UNIX operating system), gcos, os, tss, or u370], the hardware (ibm, interdata, pdp11, u3b, or VAX), and the UNIX system variant (RES, RT, or mert).

Other options are taken to be either link editor options or C-compatible object programs.

Arguments that end with .c are taken to be C language source programs; they are compiled, and each object program is left on the file whose name is that of the source with .o substituted for .c. The .o file is normally deleted.

In the same way, arguments whose names end with .s are taken to be assembly source programs and are assembled producing a .o file.

These programs, together with the results of any compilations specified, are linked (in the order given) to produce an executable program with the name **a.out**.

#### A C PROGRAM CHECKER—"lint"

##### A. General

The **lint** program examines C language source programs detecting a number of bugs and obscurities. It enforces the type rules of C language more strictly than the C compiler. It may also be used to enforce a number of portability restrictions involved in moving programs between different machines and/or operating systems. Another option detects a number of wasteful or error prone constructions which nevertheless are legal. The **lint** program accepts multiple input files and library specifications and checks them for consistency.

##### Usage

The **lint(1)** command has the form:

**lint** [options] files ... library-descriptors ...

where *options* are optional flags to control **lint** checking and messages; *files* are the files to be checked which end with .c; and *library-descriptors* are the names of libraries to be used in checking the program.

The options that are currently supported by the **lint** command are:

- a Suppress messages about assignments of long values to variables that are not long.
- b Suppress messages about break statements that cannot be reached.
- c Suppress messages about casts that have questionable portability.
- h Do not apply heuristics (which attempt to detect bugs, improve style, and reduce waste).
- n Do not check for compatibility with either the standard or the portable **lint** library.
- p Attempt to check portability to other dialects of C language (IBM and Honeywell).
- u Suppress messages about function and external variables used and not defined or defined and not used.



- v Suppress messages about unused arguments in functions.
- x Do not report variables referred to by external declarations but never used.

When more than one option is used, they should be combined into a single argument, such as, `-ab` or `-xha`.

The names of files that contain C language programs should end with the suffix `.c` which is mandatory or `lint` and the C compiler.

The `lint` program accepts certain arguments, such as:

`-ly`

These arguments specify libraries that contain functions used in the C language program. The source code is tested for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library arguments. These files all begin with the comment:

```
/* LINTLIBRARY */
```

which is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. The `VARARGS` and `ARGSUSED` comments can be used to specify features of the library functions.

The `lint` library files are processed almost exactly like ordinary source files. The only difference is that functions which are defined on a library file but are not used on a source file do not result in messages. The `lint` program does not simulate a full library search algorithm and will print messages if the source files contain a redefinition of a library routine.

By default, `lint` checks the programs it is given against a standard library file which contains descriptions of the programs which are normally loaded when a C language program is run. When the `-p` option is used, another file is checked containing descriptions of the standard I/O library routines which are expected to be portable across various machines. The `-n` option can be used to suppress all library checking.

## B. Types of Messages

The following paragraphs describe the major categories of messages printed by `lint`.

### Unused Variables and Functions

As sets of programs evolve and develop, previously used variables and arguments to functions may become unused. It is not uncommon for external variables or even entire functions to become unnecessary and yet not be removed from the source. These types of errors rarely cause working programs to fail, but are a source of inefficiency and make programs harder to understand and change. Also, information about such unused variables and functions can occasionally serve to discover bugs.

The `lint` program prints messages about variables and functions which are defined but not otherwise mentioned. An exception is variables which are declared through explicit `extern` statements but are never referenced; thus the statement

```
extern float sin();
```

will evoke no comment if `sin` is never used. Note that this agrees with the semantics of the C compiler. In some cases, these unused external declarations might be of some interest and can be discovered by using the `-x` option with the `lint` command.



Certain styles of programming require many functions to be written with similar interfaces; frequently, some of the arguments may be unused in many of the calls. The `-v` option is available to suppress the printing of messages about unused arguments. When `v` is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments. This can be considered an active (and preventable) waste of the register resources of the machine.

Messages about unused arguments can be suppressed for one function by adding the comment:

```
/* ARGSUSED */
```

to the program before the function. This has the effect of the `-v` option for only one function. Also, the comment:

```
/* VARARGS */
```

can be used to suppress messages about variable number of arguments in calls to a function. The comment should be added before the function definition. In some cases, it is desirable to check the first several arguments and leave the later arguments unchecked. This can be done with a digit giving the number of arguments which should be checked, such as:

```
/* VARARGS2 */
```

will cause only the first two arguments to be checked.

There is one case where information about unused or undefined variables is more distracting than helpful. This is when `lint` is applied to some but not all files out of a collection which are to be loaded together. In this case, many of the functions and variables defined may not be used. Conversely, many functions and variables defined elsewhere may be used. The `-u` option may be used to suppress the spurious messages which might otherwise appear.

#### Set/Used Information

The `lint` program attempts to detect cases where a variable is used before it is set. The `lint` program detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a "use", since the actual use may occur at any later time, in a data dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement since the true flow of control need not be discovered. It does mean that `lint` can print messages about some programs which are legal, but these programs would probably be considered bad on stylistic grounds. Because static and external variables are initialized to zero, no meaningful information can be discovered about their uses. The `lint` program does deal with initialized automatic variables.

The set/used information also permits recognition of those local variables which are set and never used. These form a frequent source of inefficiencies and may also be symptomatic of bugs.

#### Flow of Control

The `lint` program attempts to detect unreachable portions of the programs which it processes. It will print messages about unlabeled statements immediately following `goto`, `break`, `continue`, or `return` statements. An attempt is made to detect loops which can never be left at the bottom and recognize the special cases `while(1)` and `for(;;)` as infinite loops. The `lint` program also prints messages about loops which cannot be entered at the top. Some valid programs may have such loops which are considered to be bad style at best and bugs at worst.

The `lint` program has no way of detecting functions which are called and never return. Thus, a call to `exit` may cause an unreachable code which `lint` does not detect. The most serious effects of this are in the determination of returned function values (see "Function Values"). If a particular place in the program cannot be reached



but it is not apparent to **lint**, the comment

```
/* NOTREACHED */
```

can be added at the appropriate place. This comment will inform **lint** that a portion of the program cannot be reached.

The **lint** program will not print a message about unreachable **break** statements. Programs generated by **yacc** and especially **lex** may have hundreds of unreachable **break** statements. The **-O** option in the C compiler will often eliminate the resulting object code inefficiency. Thus, these unreachable statements are of little importance. There is typically nothing the user can do about them, and the resulting messages would clutter up the **lint** output. If these messages are desired, **lint** can be invoked with the **-b** option.

### Function Values

Sometimes functions return values which are never used. Sometimes programs incorrectly use function "values" which have never been returned. The **lint** program addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both

```
return( expr );
```

and

```
return ;
```

statements is cause for alarm; the **lint** program will give the message

```
function name contains return(e) and return
```

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
f ( a ) {  
    if ( a ) return ( 3 );  
    g ( );  
}
```

Notice that, if **a** tests false, **f** will call **g** and then return with no defined return value; this will trigger a message from **lint**. If **g**, like **exit**, never returns, the message will still be produced when in fact nothing is wrong.

In practice, some potentially serious bugs have been discovered by this feature. It also accounts for a substantial portion of the redundant messages produced by **lint**.

On a global scale, **lint** detects cases where a function returns a value; and this value is sometimes or never used. When the value is never used, it may constitute an inefficiency in the function definition. When the value is sometimes unused, it may represent bad style (e.g., not testing for error conditions).

The dual problem, using a function value when the function does not return one, is also detected. This is a serious problem.

### Type Checking

The **lint** program enforces the type checking rules of C language more strictly than the compilers do. The additional checking is in four major areas:

- Across certain binary operators and implied assignments



- At the structure selection operators
- Between the definition and uses of functions
- In the use of enumerations.

There are a number of operators which have an implied balancing between types of the operands. The assignment, conditional (?:), and relational operators have this property. The argument of a **return** statement and expressions used in initialization suffer similar conversions. In these operations, **char**, **short**, **int**, **long**, **unsigned**, **float**, and **double** types may be freely intermixed. The types of pointers must agree exactly except that arrays of *x*'s can, of course, be intermixed with pointers to *x*'s.

The type checking rules also require that, in structure references, the left operand of the  $\rightarrow$  be a pointer to structure, the left operand of the  $\cdot$  be a structure, and the right operand of these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types **float** and **double** may be freely matched, as may the types **char**, **short**, **int**, and **unsigned**. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types or other enumerations and that the only operations applied are =, initialization, ==, !=, and function arguments and return values.

If it is desired to turn off strict type checking for an expression, the comment

```
/* NOSTRICT */
```

should be added to the program immediately before the expression. This comment will prevent strict type checking for only the next line in the program.

### Type Casts

The type cast feature in C language was introduced largely as an aid to producing more portable programs. Consider the assignment

```
p = 1 ;
```

where *p* is a character pointer. The **lint** program will print a message as a result of detecting this. Consider the assignment

```
p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this and has clearly signaled his/her intentions. It seems harsh for **lint** to continue to print messages about this. On the other hand, if this code is moved to another machine, such code should be looked at carefully. The **-c** flag controls the printing of comments about casts. When **-c** is in effect, casts are treated as though they were assignments subject to messages; otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

### Nonportable Character Use

On some systems, characters are signed quantities with a range from -128 to 127. On other C language implementations, characters take on only positive values. Thus, **lint** will print messages about certain comparisons and assignments as being illegal or nonportable. For example, the fragment

```
char c;
```

```
...
```

```
if( (c = getchar()) < 0 ) ...
```



will work on one machine but will fail on machines where characters always take on positive values. The real solution is to declare `c` as an integer since `getchar` is actually returning integer values. In any case, `lint` will print the message "nonportable character comparison".

A similar issue arises with bit fields. When assignments of constant values are made to bit fields, the field may be too small to hold the value. This is especially true because on some machines bit fields are considered as signed quantities. While it may seem logical to consider that a two bit field declared of type `int` cannot hold the value 3, the problem disappears if the bit field is declared to have type `unsigned`.

#### Assignments of "longs" to "ints"

Bugs may arise from the assignment of `long` to an `int`, which will truncate the contents. This may happen in programs which have been incompletely converted to use `typedefs`. When a `typedef` variable is changed from `int` to `long`, the program can stop working because some intermediate results may be assigned to `ints`, which is truncated. Since there are a number of legitimate reasons for assigning `longs` to `ints`, the detection of these assignments is enabled by the `-a` option.

#### Strange Constructions

Several perfectly legal, but somewhat strange, constructions are detected by `lint`. The messages hopefully encourage better code quality, clearer style, and may even point out bugs. The `-h` option is used to enable these checks. For example, in the statement

```
*p++ ;
```

the `*` does nothing. This provokes the message "null effect" from `lint`. The following program fragment:

```
unsigned x ;  
if( x < 0 ) ...
```

results in a test that will never succeed. Similarly, the test

```
if( x > 0 ) ...
```

is equivalent to

```
if( x != 0 )
```

which may not be the intended action. The `lint` program will print the message "degenerate unsigned comparison" in these cases. If a program contains something similar to

```
if( 1 != 0 ) ...
```

`lint` will print the message "constant in conditional context" since the comparison of 1 with 0 gives a constant result.

Another construction detected by `lint` involves operator precedence. Bugs which arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting making such bugs extremely hard to find. For example, the statement

```
if( x&077 == 0 ) ...
```



or

```
x << 2 + 40
```

probably do not do what was intended. The best solution is to parenthesize such expressions, and `lint` encourages this by an appropriate message.

Finally, when the `-h` option has been used, `lint` prints messages about variables which are redeclared in inner blocks in a way that conflicts with their use in outer blocks. This is legal but is considered to be bad style, usually unnecessary, and frequently a bug.

### Old Syntax

Several forms of older syntax are now illegal. These fall into two classes—assignment operators and initialization.

The older forms of assignment operators (e.g., `=+`, `=-`, ...) could cause ambiguous expressions, such as:

```
a == -1 ;
```

which could be taken as either

```
a == -1 ;
```

or

```
a = -1 ;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer and preferred operators (e.g., `+=`, `-=`, ...) have no such ambiguities. To encourage the abandonment of the older forms, `lint` prints messages about these old-fashioned operators.

A similar issue arises with initialization. The older language allowed

```
int x 1 ;
```

to initialize `x` to 1. This also caused syntactic difficulties. For example, the initialization

```
int x ( -1 ) ;
```

looks somewhat like the beginning of a function declaration:

```
int x ( y ) { ...
```

and the compiler must read past `x` in order to determine the correct meaning. Again, the problem is even more perplexing when the initializer involves a macro. The current syntax places an equals sign between the variable and the initializer:

```
int x = -1 ;
```

This is free of any possible syntactic ambiguity.

### Pointer Alignment

Certain pointer assignments may be reasonable on some machines and illegal on others due entirely to alignment restrictions. For example, on the PDP-11, it is reasonable to assign integer pointers to double pointers



since double-precision values may begin on any integer boundary. On the Honeywell 6000, double-precision values must begin on even word boundaries. Thus, not all such assignments make sense. The `lint` program tries to detect cases where pointers are assigned to other pointers and such alignment problems might arise. The message "possible pointer alignment problem" results from this situation whenever either the `-p` or `-h` options are used.

#### Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine dependent. For example, on machines (like the PDP-11) in which the stack runs backwards, function arguments will probably be best evaluated from right to left. On machines with a stack running forward, left to right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators which have side effects, such as the assignment operators and the increment and decrement operators.

In order that the efficiency of C language on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler. In fact, the various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect and also used elsewhere in the same expression, the result is explicitly undefined.

The `lint` program checks for the important special case where a simple scalar variable is affected. For example, the statement

```
a[i] = b[i++];
```

will cause `lint` to print the message

```
warning: i evaluation order undefined
```

in order to call attention to this condition.

#### C. Portability

C language on the Honeywell and IBM systems is used, in part, to write system code for the host operating system. This means that the implementation of C language tends to follow local conventions rather than adhere strictly to UNIX system conventions. Despite these differences, many C language programs have been successfully moved to GCOS and the various IBM installations with little effort. This section describes some of the differences between the implementations and discusses the `lint` features which encourage portability.

A related difficulty comes from the amount of information retained about external names during the loading process. On the UNIX system, externally known names have seven significant characters with the uppercase and lowercase distinction kept. On the IBM systems, there are eight significant characters; but the case distinction is lost. On GCOS, there are only six characters of a single case. This leads to situations where programs run on the UNIX system but encounter loader problems on the IBM or GCOS systems. The `-p` option causes all external symbols to be mapped to one case and truncated to six characters, providing a worst-case analysis.

A number of differences arise in the area of character handling. Characters in the UNIX system are 8-bit ASCII, while they are 8-bit EBCDIC on the IBM and 9-bit ASCII on GCOS. Also, character strings go from high to low-bit positions "left to right" on GCOS and IBM and low to high "right to left" on the PDP-11. This means that code attempting to construct strings out of character constants or attempting to use characters as indices into arrays must be looked at with great suspicion. The `lint` program is of little help here except to flag multicharacter character constants.

Of course, the word size differs from machine to machine. This causes less trouble than might be expected at least when moving from the UNIX system (16-bit words) to the IBM (32-bits) or GCOS (36-bits). The main



problems are likely to arise in shifting or masking. The C language now supports a bit-field facility, which can be used to write much of this code in a reasonably portable way. Frequently, portability of such code can be enhanced by slight rearrangements in coding style. Many of the incompatibilities seem to have the flavor of writing

```
x &= 0177700 ;
```

to clear the low order six bits of *x*. This suffices on the PDP-11 but fails badly on GCOS and IBM. If the bit-field feature cannot be used, the same effect can be obtained by writing

```
x &= ~ 077 ;
```

which will work on all these machines.

The right shift operator is arithmetic shift on the PDP-11 and logical shift on most other machines. To obtain a logical shift on all machines, the left operand can be typed **unsigned**. Characters are considered signed integers on the PDP-11 and unsigned on the other machines. This persistence of the sign bit may be reasonably considered a bug in the PDP-11 hardware which has infiltrated itself into the C language. If there were a good way to discover the programs which would be affected, the C language could be changed. In any case, lint is no help here.

The previous discussion may have made the problem of portability seem bigger than it is. The issues involved here are rarely subtle or mysterious, at least to the implementor of the program, although they can involve some work to straighten out. The most serious bar to the portability of UNIX system utilities has been the inability to mimic essential UNIX system functions on the other systems. The inability to seek to a random character position in a text file or to establish a pipe between processes has involved far more rewriting and debugging than any of the differences in C compilers. On the other hand, lint has been very helpful in moving the UNIX operating system and associated utility programs to other machines.

## A SYMBOLIC DEBUGGING PROGRAM—"sdb"

### A. General

This part describes the symbolic debugger **sdb**(1) as implemented for C language and Fortran 77 programs on the UNIX operating system. The **sdb** program is useful both for examining "core images" of aborted programs and for providing an environment in which execution of a program can be monitored and controlled.

The **sdb** program allows interaction with a debugged program at the source language level. When debugging a core image from an aborted program, **sdb** reports which line in the source program caused the error and allows all variables to be accessed symbolically and to be displayed in the correct format.

Breakpoints may be placed at selected statements or the program may be single stepped on a line-by-line basis. To facilitate specification of lines in the program without a source listing, **sdb** provides a mechanism for examining the source text. Procedures may be called directly from the debugger. This feature is useful both for testing individual procedures and for calling user-provided routines which provided formatted printout of structured data.

### B. Usage

In order to use **sdb** to its full capabilities, it is necessary to compile the source program with the **-g** option. This causes the compiler to generate additional information about the variables and statements of the compiled program. When the **-g** option has been specified, **sdb** can be used to obtain a trace of the called functions at the time of the abort and interactively display the values of variables.



A typical sequence of **shell** commands for debugging a core image is

```
$ cc -g prgm.c -o prgm
$ prgm
Bus error - core dumped
$ sdb prgm
main:25:      x[i] = 0;
*
```

The program **prgm** was compiled with the **-g** option and then executed. An error occurred which caused a core dump. The **sdb** program is then invoked to examine the core dump to determine the cause of the error. It reports that the bus error occurred in function *main* at line 25 (line numbers are always relative to the beginning of the file) and outputs the source text of the offending line. The **sdb** program then prompts the user with an **\*** indicating that it awaits a command.

It is useful to know that **sdb** has a notion of current function and current line. In this example, they are initially set to *main* and "25", respectively.

In the above example, **sdb** was called with one argument, *prgm*. In general, it takes three arguments on the command line. The first is the name of the executable file which is to be debugged; it defaults to *a.out* when not specified. The second is the name of the core file, defaulting to *core*; and the third is the name of the directory containing the source of the program being debugged. The **sdb** program currently requires all source to reside in a single directory. The default is the working directory. In the example, the second and third arguments defaulted to the correct values, so only the first was specified.

It is possible that the error occurred in a function which was not compiled with the **-g** option. In this case, **sdb** prints the function name and the address at which the error occurred. The current line and function are set to the first executable line in *main*. The **sdb** program will print an error message if *main* was not compiled with the **-g** option, but debugging can continue for those routines compiled with the **-g** option. Figure 3.1 shows a typical example of **sdb** usage.

#### Printing a Stack Trace

It is often useful to obtain a listing of the function calls which led to the error. This is obtained with the **t** command. For example:

```
*t
sub(x=2,y=3)      [prgm.c:25]
inter(i=16012)    [prgm.c:96]
main(argc=1,argv=0x7ffff54,envp=0x7ffff5c) [prgm.c:15]
```

This indicates that the error occurred within the function *sub* at line 25 in file *prgm.c*. The *sub* function was called with the arguments *x=2* and *y=3* from *inter* at line 96. The *inter* function was called from *main* at line 15. The *main* function is always called by the **shell** with three arguments often referred to as *argc*, *argv*, and *envp*. Note that *argv* and *envp* are pointers, so their values are printed in hexadecimal.

#### Examining Variables

The **sdb** program can be used to display variables in the stopped program. Variables are displayed by typing their name followed by a slash, so

```
*errflag/
```

causes **sdb** to display the value of variable *errflag*. Unless otherwise specified, variables are assumed to be either local to or accessible from the current function. To specify a different function, use the form

```
*sub:i/
```



to display variable *i* in function *sub*. F77 users can specify a common block variable in the same manner.

The **sdb** program supports a limited form of pattern matching for variable and function names. The symbol **\*** is used to match any sequence of characters of a variable name and **?** to match any single character. Consider the following commands:

```
*x*/  
*sub:y?/  
**/
```

The first prints the values of all variables beginning with *x*, the second prints the values of all two letter variables in function *sub* beginning with *y*, and the last prints all variables. In the first and last examples, only variables accessible from the current function are printed. The command

```
**./
```

displays the variables for each function on the call stack.

The **sdb** program normally displays the variable in a format determined by its type as declared in the source program. To request a different format, a specifier is placed after the slash. The specifier consists of an optional length specification followed by the format. The length specifiers are:

|   |                         |
|---|-------------------------|
| b | One byte                |
| h | Two bytes (half word)   |
| l | Four bytes (long word). |

The lengths are only effective with the formats **d**, **o**, **x**, and **u**. If no length is specified, the word length of the host machine is used. A numeric length specifier may be used for the **s** or **a** commands. These commands normally print characters until either a null is reached or 128 characters are printed. The number specifies how many characters should be printed. There are a number of format specifiers available:

|   |                                                                                  |
|---|----------------------------------------------------------------------------------|
| c | Character                                                                        |
| d | Decimal                                                                          |
| u | Decimal unsigned                                                                 |
| o | Octal                                                                            |
| x | Hexadecimal                                                                      |
| f | 32-bit single-precision floating point                                           |
| g | 64-bit double-precision floating point                                           |
| s | Assume variable is a string pointer and print characters until a null is reached |
| a | Print characters starting at the variable's address until a null is reached      |
| p | Pointer to function                                                              |
| i | Interpret as a machine-language with addresses printed symbolically              |
| I | Interpret as a machine-language with addresses printed symbolically.             |



For example, the variable *i* can be displayed with

```
*i/x
```

which prints out the value of *i* in hexadecimal.

The *sdb* program also knows about structures, arrays, and pointers so that all of the following commands work.

```
*array[2][3]/
*sym.id/
*psym->usage/
*xsym[20].p->usage/
```

The only restriction is that array subscripts must be numbers. Depending on your machine, accessing arrays may be limited to 1-dimensional arrays. Note that as a special case:

```
*psym->/d
```

displays the location pointed to by *psym* in decimal.

Core locations can also be displayed by specifying their absolute addresses. The command

```
*1024/
```

displays location 1024 in decimal. As in C language, numbers may also be specified in octal or hexadecimal so the above command is equivalent to both

```
*02000/
*0x400/
```

It is possible to mix numbers and variables so that

```
*1000.x/
```

refers to an element of a structure starting at address 1000, and

```
*1000->x/
```

refers to an element of a structure whose address is at 1000. For commands of the type *\*1000.x/* and *\*1000->x/*, the *sdb* program uses the structure template of the last structured referenced.

The address of a variable is printed with the = p, so

```
*i=
```

displays the address of *i*. Another feature whose usefulness will become apparent later is the command

```
*./
```

which displays the last variable typed again.

### C. Source File Display and Manipulation

The *sdb* program has been designed to make it easy to debug a program without constant reference to a current source listing. Facilities are provided which perform context searches within the source files of the program being debugged and to display selected portions of the source files. The commands are similar to those



of the UNIX system text editor `ed(1)`. Like the editor, `sdb` has a notion of current file and line within the file. The `sdb` program also knows how the lines of a file are partitioned into functions, so it also has a notion of current function. As noted in other parts of this document, the current function is used by a number of `sdb` commands.

#### Displaying the Source File

Four commands exist for displaying lines in the source file. They are useful for perusing the source program and for determining the context of the current line. The commands are:

- `p` Prints the current line.
- `w` Window; prints a window of ten lines around the current line.
- `z` Prints ten lines starting at the current line. Advances the current line by ten.
- `control-d` Scrolls; prints the next ten lines and advances the current line by ten. This command is used to cleanly display long segments of the program.

When a line from a file is printed, it is preceded by its line number. This not only gives an indication of its relative position in the file but is also used as input by some `sdb` commands.

#### Changing the Current Source File or Function

The `e` command is used to change the current source file. Either of the forms

- `*e function`
- `*e file.c`

may be used. The first causes the file containing the named function to become the current file, and the current line becomes the first line of the function. The other form causes the named file to become current. In this case, the current line is set to the first line of the named file. Finally, an `e` command with no argument causes the current function and file named to be printed.

#### Changing the Current Line in the Source File

The `z` and `control-d` commands have a side effect of changing the current line in the source file. The following paragraphs describe other commands that change the current line.

There are two commands for searching for instances of regular expressions in source files. They are

- `*/regular expression/`
- `*?regular expression?`

The first command searches forward through the file for a line containing a string that matches the regular expression and the second searches backwards. The trailing `/` and `?` may be omitted from these commands. Regular expression matching is identical to that of `ed(1)`.

The `+` and `-` commands may be used to move the current line forwards or backwards by a specified number of lines. Typing a new-line advances the current line by one, and typing a number causes that line to become the current line in the file. These commands may be combined with the display commands so that

- `*+15z`



advances the current line by 15 and then prints 10 lines.

#### D. A Controlled Environment for Program Testing

One very useful feature of **sdb** is breakpoint debugging. After entering **sdb**, certain lines in the source program may be specified to be *breakpoints*. The program is then started with a **sdb** command. Execution of the program proceeds as normal until it is about to execute one of the lines at which a breakpoint has been set. The program stops and **sdb** reports the breakpoint where the program stopped. Now, **sdb** commands may be used to display the trace of function calls and the values of variables. If the user is satisfied the program is working correctly to this point, some breakpoints can be deleted and others set; and then program execution may be continued from the point where it stopped.

A useful alternative to setting breakpoints is single stepping. The **sdb** program can be requested to execute the next line of the program and then stop. This feature is especially useful for testing new programs, so they can be verified on a statement-by-statement basis. Note that if an attempt is made to single step through a function which has not been compiled with the **-g** option execution proceeds until a statement in a function compiled with the **-g** option is reached. It is also possible to have the program execute one machine level instruction at a time. This is particularly useful when the program has not been compiled with the **-g** option.

#### Setting and Deleting Breakpoints

Breakpoints can be set at any line in a function which contains executable code. The command format is:

```
*12b
*proc:12b
*proc:b
*b
```

The first form sets a breakpoint at line 12 in the current file. The line numbers are relative to the beginning of the file as printed by the source file display commands. The second form sets a breakpoint at line 12 of function *proc*, and the third sets a breakpoint at the first line of *proc*. The last sets a breakpoint at the current line.

Breakpoints are deleted similarly with the commands

```
*12d
*proc:12d
*proc:d
```

In addition, if the command **d** is given alone, the breakpoints are deleted interactively. Each breakpoint location is printed, and a line is read from the user. If the line begins with a **y** or **d**, the breakpoint is deleted.

A list of the current breakpoints is printed in response to a **B** command, and the **D** command deletes all breakpoints. It is sometimes desirable to have **sdb** automatically perform a sequence of commands at a breakpoint and then have execution continue. This is achieved with another form of the **b** command

```
*12b t;x/
```

causes both a trace back and the value of *x* to be printed each time execution gets to line 12. The **a** command is a variation of the above command. There are two forms:

```
*proc:a
*proc:12a
```

The first prints the function name and its arguments each time it is called, and the second prints the source line each time it is about to be executed. For both forms of the **a** command, execution continues after the function name or source line is printed.



### Running the Program

The **r** command is used to begin program execution. It restarts the program as if it were invoked from the shell. The command

```
*r args
```

runs the program with the given arguments as if they had been typed on the **shell** command line. If no arguments are specified, then the arguments from the last execution of the program are used. To run a program with no arguments, use the **R** command.

After the program is started, execution continues until a breakpoint is encountered, a signal such as **INTERUPT** or **QUIT** occurs, or the program terminates. In all cases after an appropriate message is printed, control returns to **sdb**.

The **c** command may be used to continue execution of a stopped program. A line number may be specified, as in:

```
*proc:12c
```

This places a temporary breakpoint at the named line. The breakpoint is deleted when the **c** command finishes. There is also a **C** command which continues but passes the signal which stopped the program back to the program. This is useful for testing user-written signal handlers. Execution may be continued at a specified line with the **g** command. For example:

```
*17 g
```

continues at line 17 of the current function. A use for this command is to avoid executing a section of code which is known to be bad. The user should not attempt to continue execution in a function different than that of the breakpoint.

The **s** command is used to run the program for a single line. It is useful for slowly executing the program to examine its behavior in detail. An important alternative is the **S** command. This command is like the **s** command but does not stop within called functions. It is often used when one is confident that the called function works correctly but is interested in testing the calling routine.

The **i** command is used to run the program one machine level instruction at a time while ignoring the signal which stopped the program. Its uses are similar to the **s** command. There is also an **I** command which causes the program to execute one machine level instruction at a time, but also passes the signal which stopped the program back to the program.

### Calling Functions

It is possible to call any of the functions of the program from **sdb**. This feature is useful both for testing individual functions with different arguments and for calling a function which prints structured data in a nice way. There are two ways to call a function:

```
*proc(arg1, arg2, ...)  
*proc(arg1, arg2, ...)/m
```

The first simply executes the function. The second is intended for calling functions (it executes the function and prints the value that it returns). The value is printed in decimal unless some other format is specified by *m*. Arguments to functions may be integer, character or string constants, or values of variables which are accessible from the current function.



An unfortunate bug in the current implementation is that if a function is called when the program is *not* stopped at a breakpoint (such as when a core image is being debugged) all variables are initialized before the function is started. This makes it impossible to use a function which formats data from a dump.

#### E. Machine Language Debugging

The **sdb** program has facilities for examining programs at the machine language level. It is possible to print the machine language statements associated with a line in the source and to place breakpoints at arbitrary addresses. The **sdb** program can also be used to display or modify the contents of the machine registers.

##### Displaying Machine Language Statements

To display the machine language statements associated with line 25 in function *main*, use the command

```
*main:25?
```

The **?** command is identical to the **/** command except that it displays from text space. The default format for printing text space is the **i** format which interprets the machine language instruction. The **control-d** command may be used to print the next ten instructions.

Absolute addresses may be specified instead of line numbers by appending a **:** to them so that

```
*0x1024:?
```

displays the contents of address *0x1024* in text space. Note that the command

```
*0x1024?
```

displays the instruction corresponding to line *0x1024* in the current function. It is also possible to set or delete a breakpoint by specifying its absolute address;

```
*0x1024:b
```

sets a breakpoint at address *0x1024*.

##### Manipulating Registers

The **x** command prints the values of all the registers. Also, individual registers may be named instead of variables by appending a **%** to their name so that

```
*r3%
```

displays the value of register *r3*.

#### F. Other Commands

To exit **sdb**, use the **q** command.

The **!** command is identical to that in **ed(1)** and is used to have the **shell** execute a command.

It is possible to change the values of variables when the program is stopped at a breakpoint. This is done with the command

```
*variable!value
```



which sets the variable to the given value. The value may be a number, character constant, register, or the name of another variable. If the variable is of type float or double, the value can also be a floating-point constant.

```
$ cat testdiv2.c
main(argc, argv, envp)
char **argv, **envp; {
    int i;
    i = div2(-1);
    printf( "-1/2 = %d\n" , i);
}
div2(i) {
    int j;
    j = i>>1;
    return(j);
}
$ cc -g testdiv2.c
$ a.out
-1/2 = -1
$ sdb
No core image          # Warning message from sdb
*/^div2                # Search for function "div2"
7: div2(i) {            # It starts on line 7
*z                     # Print the next few lines
7: div2(i) {
8:     int j;
9:     j = i>>1;
10:    return(j);
11: }
*div2:b                # Place a breakpoint at the beginning of "div2"
div2:9 b               # Sdb echoes proc name and line number
*r                     # Run the function
a.out                  # Sdb echoes command line executed
Breakpoint at          # Executions stops just before line 9
div2:9:    j = i>>1;
*t                     # Print trace of subroutine calls
div2(i=-1) [testdiv2.c:9]
main(argc=1,argv=0x7fffff50,envp=0x7fffff58) [testdiv2.c:4]
*i/                    # Print i
-1
*s                     # Single step
div2:10:    return(j);  # Execution stops just before line 10
*j/            # Print j
-1
*9d             # Delete the breakpoint
*div2(1)/       # Try running "div2" with different arguments
0
*div2(-2)/
-1
*div2(-3)/
-2
*q
$
```

Fig. 3.1—Example of sdb Usage



## 4. FORTRAN

### INTRODUCTION

This section describes the implementation of the Fortran programming language on the UNIX operating system. This section contains the following parts:

- **FORTTRAN 77**—Describes the implementation of Fortran 77 on the system in terms of the variations from the American National Standard.
- **A RATIONAL FORTRAN PREPROCESSOR—"ratfor"**—A preprocessor which provides a means for writing Fortran in a fashion similar to C language. This preprocessor provides (among other things) simplified control-flow statements.

Both parts assume that the user is already familiar with Fortran 77.

Throughout this section, each reference of the form **name(1M)**, **name(7)**, or **name(8)** refers to entries in the UNIX System Administrator's Manual. All other references to entries of the form **name(N)**, where "N" is a number (1 through 6) possibly followed by a letter, refer to entry **name** in section N of the UNIX System User's Manual.

### FORTTRAN 77

#### A. General

This part describes the compiler and run-time system for Fortran 77 as implemented on the UNIX operating system. Fortran 77 became an official American National Standard on April 3, 1978. The implementation of Fortran 77 on the UNIX operating system varies from the American National Standard. This document describes the difference between the American National Standard and the current implementation. Also, this document describes the interfaces between procedures and the file formats assumed by the I/O system.

#### Usage

The command to run the compiler is

**f77** options file

The **f77(1)** command is a general purpose command for compiling and loading Fortran and Fortran-related files. Ratfor source files will be preprocessed before being presented to the Fortran compiler. C language and assembler source files will be compiled by the appropriate programs. Object files will be loaded. [The **f77(1)** and **cc(1)** commands cause slightly different loading sequences to be generated since Fortran programs need a few extra libraries and a different startup routine than do C language programs.] The following file name suffixes are understood:

|           |                        |
|-----------|------------------------|
| <b>.f</b> | Fortran source file    |
| <b>.r</b> | Ratfor source file     |
| <b>.c</b> | C language source file |
| <b>.s</b> | Assembler source file  |
| <b>.o</b> | Object file            |



The following flags are understood:

- S           Generate assembler output for each source file, but do not assemble it. Assembler output for a source file **x.f**, **x.r**, or **x.c** is put on file **x.s**.
- c           Compile but do not load. Output for **x.f**, **x.r**, **x.c**, or **x.s** is put on file **x.o**.
- m           Apply the M4 macro preprocessor to each Ratfor source file before using the appropriate compiler.
- f           Apply the Ratfor processor to all relevant files and leave the output from **x.r** on **x.f**. Do not compile the resulting Fortran program.
- p           Generate code to produce usage profiles.
- of           Put executable module on file **f**. (Default is **a.out**).
- w           Suppress all warning messages.
- w66          Suppress warnings about Fortran 66 features used.
- O           Invoke the C language object code optimizer.
- C           Compile code the checks that subscripts are within array bounds.
- onetrip      Compile code that performs every **do** loop at least once.
- U           Do not convert uppercase letters to lowercase. The default is to convert Fortran programs to lowercase.
- u           Make the default type of a variable **undefined**.
- I2           On machines which support short integers, make the default integer constants and variables short. (-I4 is the standard value of this option.) All logical quantities will be short.
- R           The remaining characters in the argument are used as a Ratfor flag argument.
- F           Ratfor and source programs are preprocessed into Fortran files, but those files are not compiled or removed.

All library names (arguments beginning **-l**) and other options not ending with one of the special suffixes are passed to the loader. See f77(1) for additional options.

#### B. Language Extensions

Fortran 77 includes almost all of Fortran 66 as a subset. The most important additions are a character string data type, file-oriented input/output statements, and random access I/O. Also, the language has been cleaned up considerably.

In addition to implementing the language specified in the new Fortran 77 American National Standard, this compiler implements a few extensions. Most are useful additions to the language. The remainder are extensions to make it easier to communicate with C language procedures or to permit compilation of old (1966 Standard Fortran) programs.



### Double Complex Data Type

The data type double complex is added. Each datum is represented by a pair of double-precision real variables. A double complex version of every complex built-in function is provided. The specific function names begin with *z* instead of *c*.

### Internal Files

The Fortran 77 American National Standard introduces *internal files* (memory arrays) but restricts their use to formatted sequential I/O statements. This I/O system also permits internal files to be used in direct and unformatted reads and writes.

### Implicit Undefined Statement

Fortran has a fixed rule that the type of a variable that does not appear in a type statement is integer if its first letter is *i*, *j*, *k*, *l*, *m* or *n*. Otherwise, it is real. Fortran 77 has an implicit statement for overriding this rule. An additional type statement, *undefined*, is permitted. The statement

implicit undefined(a-z)

turns off the automatic data typing mechanism, and the compiler will issue a diagnostic for each variable that is used but does not appear in a type statement. Specifying the *-u* compiler option is equivalent to beginning each procedure with this statement.

### Recursion

Procedures may call themselves directly or through a chain of other procedures.

### Automatic Storage

Two new keywords recognized are *static* and *automatic*. These keywords may appear as "types" in type statements and in *implicit* statements. Local variables are static by default; there is exactly one copy of the datum, and its value is retained between calls. There is one copy of each variable declared *automatic* for each invocation of the procedure. Automatic variables may not appear in *equivalence*, *data*, or *save* statements.

### Variable Length Input Lines

The Fortran 77 American National Standard expects input to the compiler to be in a 72-column format: except in comment lines, the first five characters are the statement number, the next is the continuation character, and the next 66 are the body of the line. (If there are fewer than 72 characters on a line, the compiler pads it with blanks; characters after the first 72 are ignored.) In order to make it easier to type Fortran programs, this compiler also accepts input in variable length lines. An ampersand (&) in the first position of a line indicates a continuation line; the remaining characters form the body of the line. A tab character in one of the first six positions of a line signals the end of the statement number and continuation part of the line; the remaining characters form the body of the line. A tab elsewhere on the line is treated as another kind of blank by the compiler.

In the Fortran 77 Standard, there are only 26 letters—Fortran is a 1-case language. Consistent with ordinary system usage, the new compiler expects lowercase input. By default, the compiler converts all uppercase characters to lowercase except those inside character constants. However, if the *-U* compiler option is specified, uppercase letters are not transformed. In this mode, it is possible to specify external names with uppercase letters in them and to have distinct variables differing only in case. Regardless of the setting of the option, keywords will only be recognized in lowercase.



### Include Statement

The statement

```
include "stuff"
```

is replaced by the contents of the file *stuff*. Includes may be nested to a reasonable depth, currently ten.

### Binary Initialization Constants

A **logical**, **real**, or **integer** variable may be initialized in a data statement by a binary constant denoted by a letter followed by a quoted string. If the letter is *b*, the string is **binary**, and only zeroes and ones are permitted. If the letter is *o*, the string is **octal** with digits *zero* through *seven*. If the letter is *z* or *x*, the string is **hexadecimal** with digits *zero* through *nine*, *a* through *f*. Thus, the statements

```
integer a(3)  
data a/b'1010',o'12',z'a'/
```

initialize all three elements of a to ten.

### Character Strings

For compatibility with C language usage, the following backslash escapes are recognized:

|                 |                                              |
|-----------------|----------------------------------------------|
| <code>\n</code> | new-line                                     |
| <code>\t</code> | tab                                          |
| <code>\b</code> | backspace                                    |
| <code>\f</code> | form feed                                    |
| <code>\0</code> | null                                         |
| <code>\'</code> | apostrophe (does not terminate a string)     |
| <code>\"</code> | quotation mark (does not terminate a string) |
| <code>\\</code> | <code>\</code>                               |
| <code>\x</code> | where <i>x</i> is any other character.       |

Fortran 77 only has one quoting character—the apostrophe (`'`). This compiler and I/O system recognize both the apostrophe and the double quote (`"`). If a string begins with one variety of quote mark, the other may be embedded within it without using the repeated quote or backslash escapes.

Every unequivocal scalar local character variable and every character string constant is aligned on an integer word boundary. Each character string constant appearing outside a data statement is followed by a null character to ease communication with C language routines.

### Hollerith

Fortran 77 does not have the old Hollerith (**nh**) notation though the new Standard recommends implementing the old Hollerith feature in order to improve compatibility with old programs. In this compiler, Hollerith



data may be used in place of character string constants and may also be used to initialize noncharacter variables in data statements.

#### Equivalence Statements

This compiler permits single subscripts in **equivalence** statements under the interpretation that all missing subscripts are equal to 1. A warning message is printed for each such incomplete subscript.

#### One-Trip DO Loops

The Fortran 77 American National Standard requires that the range of a **do** loop not be performed if the initial value is already past the limit value, as in

```
do 10 i = 2, 1
```

The 1966 Standard stated that the effect of such a statement was undefined, but it was common practice that the range of a **do** loop would be performed at least once. In order to accommodate old programs though they were in violation of the 1966 Standard, the **-onetrip** compiler option causes nonstandard loops to be generated.

#### Commas in Formatted Input

The I/O system attempts to be more lenient than the Fortran 77 American National Standard when it seems worthwhile. When doing a formatted read of noncharacter variables, commas may be used as value separators in the input record overriding the field lengths given in the format statement. Thus, the format

```
(i10, f20.10, i4)
```

will read the record

```
-345,.05e-3,12
```

correctly.

#### Short Integers

On machines that support half word integers, the compiler accepts declarations of type **integer\*2**. (Ordinary integers follow the Fortran rules about occupying the same space as a **REAL** variable; they are assumed to be of C language type **long int**; half word integers are of C language type **short int**.) An expression involving only objects of type **integer\*2** is of that type. Generic functions return short or long integers depending on the actual types of their arguments. If a procedure is compiled using the **-I2** flag, all small integer constants will be of type **integer\*2**. If the precision of an integer-valued intrinsic function is not determined by the generic function rules, one will be chosen that returns the prevailing length (**integer\*2** when the **-I2** command flag is in effect). When the **-I2** option is in effect, all quantities of type **logical** will be short. Note that these short integer and logical quantities do not obey the standard rules for storage association.

#### Additional Intrinsic Functions

This compiler supports all of the intrinsic functions specified in the Fortran 77 Standard. In addition, there are functions for performing bitwise Boolean operations (**or**, **and**, **xor**, and **not**) and for accessing the command arguments (**getarg** and **iargc**).

#### C. Violations of the Standard

The following paragraphs describe only three known ways in which the UNIX system implementation of Fortran 77 violates the new American National Standard.



### Double Precision Alignment

The Fortran 77 American National Standard permits **common** or **equivalence** statements to force a double precision quantity onto an odd word boundary, as in the following example:

```
real a(4)
double precision b,c
equivalence (a(1),b), (a(4),c)
```

Some machines (e.g., Honeywell 6000, IBM 360) require that double precision quantities be on double word boundaries; other machines (e.g., IBM 370) run inefficiently if this alignment rule is not observed. It is possible to tell which equivalenced and common variables suffer from a forced odd alignment, but every double-precision argument would have to be assumed on a bad boundary. To load such a quantity on some machines, it would be necessary to use two separate operations. The first operation would be to move the upper and lower halves into the halves of an aligned temporary. The second would be to load that double-precision temporary. The reverse would be needed to store a result. All double-precision real and complex quantities are required to fall on even word boundaries on machines with corresponding hardware requirements and to issue a diagnostic if the source code demands a violation of the rule.

### Dummy Procedure Arguments

If any argument of a procedure is of type character, all dummy procedure arguments of that procedure must be declared in an **external** statement. This requirement arises as a subtle corollary of the way we represent character string arguments. This requirement arises because of the way character string arguments are represented and of the 1-pass nature of the compiler. A warning is printed if a dummy procedure is not declared **external**. Code is correct if there are no **character** arguments.

### T and TL Formats

The implementation of the **t** (absolute tab) and **tl** (leftward tab) format codes is defective. These codes allow rereading or rewriting part of the record which has already been processed. The implementation uses "seeks"; so if the unit is not one which allows seeks (such as a terminal) the program is in error. A benefit of the implementation chosen is that there is no upper limit on the length of a record nor is it necessary to predeclare any record lengths except where specifically required by Fortran or the operating system.

### D. Interprocedure Interface

To be able to write C language procedures that call or are called by Fortran procedures, it is necessary to know the conventions for procedure names, data representation, return values, and argument lists that the compiled code obeys.

### Procedure Names

On UNIX systems, the name of a common block or a Fortran procedure has an underscore appended to it by the compiler to distinguish it from a C language procedure or external variable with the same user-assigned name. Fortran library procedure names have embedded underscores to avoid clashes with user-assigned subroutine names.

### Data Representations

The following is a table of corresponding Fortran and C language declarations:

| Fortran     | C Language   |
|-------------|--------------|
| integer*2 x | short int x; |



|                    |                              |
|--------------------|------------------------------|
| integer x          | long int x;                  |
| logical x          | long int x;                  |
| real x             | float x;                     |
| double precision x | double x;                    |
| complex x          | struct { float r,i; } x;     |
| double complex x   | struct { double dr, di; } x; |
| character*6 x      | char x[6];                   |

By the rules of Fortran, **integer**, **logical**, and **real** data occupy the same amount of memory.

#### Return Values

A function of type **integer**, **logical**, **real**, or **double precision** declared as a C language function returns the corresponding type. A **complex** or **double complex** function is equivalent to a C language routine with an additional initial argument that points to the place where the return value is to be stored. Thus, the following:

```
complex function f( ... )
```

is equivalent to

```
f_(temp, ...)
struct { float r, i; } *temp;
...
```

A character-valued function is equivalent to a C language routine with two extra initial arguments—a data address and a length. Thus,

```
character*15 function g( ... )
```

is equivalent to

```
g_(result, length, ...)
char result[ ];
long int length;
...
```

and could be invoked in C language by

```
char chars[15];
...
g_(chars, 15L, ... );
```

Subroutines are invoked as if they were **integer**-valued functions whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the function but are used to do an indexed branch in the calling procedure. (If the subroutine has no entry points with alternate return arguments, the returned value is undefined.) The statement

```
call nret(*1, *2, *3)
```



is treated exactly as if it were the computed `goto`

```
goto (1, 2, 3), nret( )
```

#### Argument Lists

All Fortran arguments are passed by address. In addition, for every argument that is of type character or that is a dummy procedure, an argument giving the length of the value is passed. (The string lengths are long int quantities passed by value.) The order of arguments is then:

- Extra arguments for complex and character functions
- Address for each datum or function
- A long int for each character or procedure argument

Thus, the call in

```
external f
character*7 s
integer b(3)
...
call sam(f, b(2), s)
```

is equivalent to that in

```
int f( );
char s[7];
long int b[3];
...
sam_(f, &b[1], s, 0L, 7L);
```

Note that the first element of a C language array always has subscript 0, but Fortran arrays begin at 1 by default. Fortran arrays are stored in column-major order; C language arrays are stored in row-major order.

#### E. File Formats

##### Structure of Fortran Files

Fortran requires four kinds of external files: *sequential formatted* and *unformatted*, and *direct formatted* and *unformatted*. On UNIX systems, these are all implemented as ordinary files which are assumed to have the proper internal structure.

Fortran I/O is based on "records". When a direct file is opened in a Fortran program, the record length of the records must be given; and this is used by the Fortran I/O system to make the file look as if it is made up of records of the given length. In the special case that the record length is given as 1, the files are not considered to be divided into records but are treated as byte-addressable byte strings; i.e., as ordinary files on the UNIX system. (A read or write request on such a file keeps consuming bytes until satisfied rather than being restricted to a single record.)

The peculiar requirements on sequential unformatted files make it unlikely that they will ever be read or written by any means except Fortran I/O statements. Each record is preceded and followed by an integer containing the record's length in bytes.

The Fortran I/O system breaks sequential formatted files into records while reading by using each new-line as a record separator. The result of reading off the end of a record is undefined according to the Fortran 77



American National Standard. The I/O system is permissive and treats the record as being extended by blanks. On output, the I/O system will write a new-line at the end of each record. It is also possible for programs to write new-lines for themselves. This is an error, but the only effect will be that the single record the user thought was written will be treated as more than one record when being read or backspaced over.

#### Preconnected Files and File Positions

Units 5, 6, and 0 are preconnected when the program starts. Unit 5 is connected to the standard input, unit 6 is connected to the standard output, and unit 0 is connected to the standard error unit. All are connected for sequential formatted I/O.

All the other units are also preconnected when execution begins. Unit *n* is connected to a file named **fort.n**. These files need not exist nor will they be created unless their units are used without first executing an **open**. The default connection is for sequential formatted I/O.

The Fortran 77 Standard does not specify where a file which has been explicitly **opened** for sequential I/O is initially positioned. In fact, the I/O system attempts to position the file at the end. A **write** will append to the file and a **read** will result in an "end of file" indication. To position a file to its beginning, use a **rewind** statement. The preconnected units 0, 5, and 6 are positioned as they come from the parent process.

#### A RATIONAL FORTRAN PREPROCESSOR—"ratfor"

##### A. General

This part describes the **ratfor(1)** preprocessor. It is assumed that the user is familiar with the current implementation of Fortran 77 on the UNIX operating system.

The **ratfor** language allows users to write Fortran programs in a fashion similar to C language. The **ratfor** program is implemented as a preprocessor that translates this "simplified" language into Fortran. The facilities provided by **ratfor** are:

- statement grouping
- if-else and switch for decision making
- while, for, do, and repeat-until for looping
- break and next for controlling loop exits
- free form input such as multiple statements/lines and automatic continuation
- simple comment convention
- translation of **>**, **>=**, etc., into **.gt.**, **.ge.**, etc.
- return statement for functions
- define statement for symbolic parameters
- include statement for including source files.

##### B. Usage

The **ratfor** program takes either a list of file names or the standard input and writes Fortran on the standard output. Options include **-6x**, which uses **x** as a continuation character in column 6 (the UNIX system uses **&** in column 1), and **-C**, which causes **ratfor** comments to be copied into the generated Fortran.



The program `rc(1M)` provides an interface to the `ratfor(1)` command. This command is similar to `cc(1)`. Thus:

`rc options files`

compiles the files specified by *files*. Files with names ending in `.r` are `ratfor` source; other files are assumed to be for the loader. The options `-C` and `-6x` described above are recognized, as are

- `-c`     Compile only; don't load
- `-f`     Save intermediate Fortran `.f` files
- `-r`     Ratfor only; implies `-c` and `-f`
- `-2`     Use big Fortran compiler (for large programs)
- `-U`     Flag undeclared variables (not universally available).

Other options are passed on to the loader.

### C. Statement Grouping

Fortran provides no way to group statements together short of making them into a subroutine. The `ratfor` language does provide a statement grouping facility. A group of statements can be treated as a unit by enclosing them in the braces `{` and `}`. For example, the `ratfor` code

```
if (x > 100)
    { call error("x>100"); err = 1; return }
```

will be translated by the `ratfor` preprocessor into Fortran equivalent to

```
      if (x .le. 100) goto 10
      call error(5hx>100)
      err = 1
      return
10    ...
```

which should simplify programming effort. By using `{` and `}`, a group of statements can be used instead of a single statement.

Also note in the previous `ratfor` example that the character `>` was used instead of `.GT.` in the `if` statement. The `ratfor` preprocessor translates this C language type operator to the appropriate Fortran operator. More on relationship operators later.

In addition, many Fortran compilers permit character strings in quotes (like `"x>100"`). Quotes are not allowed in ANSI Fortran, so `ratfor` converts it into the right number of `Hs`.

The `ratfor` language is free form. Statements may appear anywhere on a line, and several may appear on one line if they are separated by semicolons. The previous example could also be written as

```
if (x > 100) {
    call error("x>100")
    err = 1
    return
}
```



which shows grouped statements spread over several lines. In this case, no semicolon is needed at the end of each line because **ratfor** assumes there is one statement per line unless told otherwise.

Of course, if the statement that follows the **if** is a single statement, no braces are needed.

#### D. The "if-else" Construction

The **ratfor** language provides an **else** statement. The syntax of the if-else construction is:

```
if (legal Fortran condition)
    ratfor statement
else
    ratfor statement
```

where the **else** part is optional. The legal Fortran condition is anything that can legally go into a Fortran Logical IF statement. The **ratfor** preprocessor does not check this clause since it does not know enough Fortran to know what is permitted. The ratfor statement is any **ratfor** or Fortran statement or any collection of them in braces. For example,

```
if (a <= b)
    { sw = 0; write(6, 1) a, b }
else
    { sw = 1; write(6, 1) b, a }
```

is a valid **ratfor if-else** construction. This writes out the smaller of a and b, then the larger, and sets sw appropriately.

As before, if the statement following an **if** or an **else** is a single statement, no braces are needed.

#### Nested "if" Statements

The statement that follows an **if** or an **else** can be any **ratfor** statement including another **if** or **else** statement. In general, the structure

```
if (condition) action
else if (condition) action
else action
```

provides a way to write a multibranch in **ratfor**. (The **ratfor** language also provides a **switch** statement which could be used instead, under certain conditions.) The last **else** handles the "default" condition. If there is no default action, this final **else** can be omitted. Thus, only the actions associated with the valid condition are performed. For example:

```
if (x < 0)
    x = 0
else if (x > 100)
    x = 100
```

will ensure that **x** is not less than 0 and not greater than 100.

Nested **if** and **else** statements could result in ambiguous code. In **ratfor** when there is more **if** statements than **else** statements, **else** statements are associated with the closest previous **if** statement that currently does



not have an associated **else** statement. For example:

```
if (x > 0)
if(y > 0)
write(6,1) x, y
else
write(6,2) y
```

is interpreted by the **ratfor** preprocessor as

```
if (x > 0) {
    if (y > 0)
        write(6, 1) x, y
    else
        write(6, 2) y
}
```

in which the braces are assumed. If the other association is desired, it must be written as

```
if (x > 0) {
    if (y > 0)
        write(6, 1) x, y
}
else
    write(6, 2) y
```

with the braces specified.

#### E. The "switch" Statement

The **switch** statement provides a way to express multiway branches which branch on the value of some integer-valued expression. The syntax is

```
switch (expression) {
    case expr1:
        statements
    case expr2, expr3:
        statements
    ...
    default:
        statements
}
```

where each **case** is followed by an integer expression (or several integer expressions separated by commas). The **switch expression** is compared to each **case expr** until a match is found. Then the *statements* following that **case** are executed. If no **cases** match *expression*, then the *statements* following **default** are executed. The **default** section of a **switch** is optional.

When the *statements* associated with a **case** is executed, the entire **switch** is exited immediately. This is somewhat different than C language.

#### F. The "do" Statement

The **do** statement in **ratfor** is quite similar to the **DO** statement in Fortran except that it uses no statement number (braces are used to mark the end of the **do** instead of a statement number). The syntax of the **ratfor**



do statement is

```
do legal-Fortran-DO-text {
    ratfor statements
}
```

The *legal-Fortran-DO-text* must be something that can legally be used in a Fortran DO statement. Thus if a local version of Fortran allows DO limits to be expressions (which is not currently permitted in ANSI Fortran), they can be used in a **ratfor do** statement. The *ratfor statements* are enclosed in braces; but as with the **if**, a single statement need not have braces around it. For example, the following code sets an array to 0:

```
do i = 1, n
    x(i) = 0.0
```

and the code

```
do i = 1, n
    do j = 1, n
        m(i, j) = 0
```

sets the entire array *m* to zero.

#### G. The "break" and "next" Statements

The **ratfor break** and **next** statements provide a means for leaving a loop early and one for beginning the next iteration. The **break** causes an immediate exit from the **do**; in effect, it is a branch to the statement *after* the **do**. The **next** is a branch to the bottom of the loop, so it causes the next iteration to be done. For example, this code skips over negative values in an array

```
do i = 1, n {
    if (x(i) < 0.0)
        next
    process positive element
}
```

The **break** and **next** statements will also work in the other **ratfor** looping constructions and will be discussed with each looping construction.

The **break** and **next** can be followed by an integer to indicate breaking or iterating that level of enclosing loop. For example:

```
break 2
```

exits from two levels of enclosing loops, and

```
break 1
```

is equivalent to **break**. The

```
next 2
```

iterates the second enclosing loop.



#### H. The "while" Statement

The **ratfor** language provides a **while** statement. The syntax of the **while** statement is

```
while (legal-Fortran-condition)  
  ratfor statement
```

As with the **if**, legal-Fortran-condition is something that can go into a Fortran Logical IF, and ratfor statement is a single statement which may be multiple statements enclosed in braces.

For example, suppose nextch is a function which returns the next input character both as a function value and in its argument. Then a **while** loop to find the first nonblank character could be

```
while (nextch(ich) == iblank)  
  ;
```

where a semicolon by itself is a null statement (which is necessary here to mark the end of the **while**). If the semicolon were not present, the **while** would control the next statement. When the loop is exited, ich contains the first nonblank.

#### I. The "for" Statement

The **for** statement is another **ratfor** loop. A **for** statement allows explicit initialization and increment steps as part of the statement.

The syntax of the **for** statement is

```
for ( init ; condition ; increment )  
  ratfor statement
```

where *init* is any single Fortran statement which is executed once before the loop begins. The increment is any single Fortran statement that is executed at the end of each pass through the loop before the test. The condition is again anything that is legal in a Fortran Logical IF. Any of init, condition, and increment may be omitted although the semicolons must always be present. A nonexistent condition is treated as always true, so

```
for (;;)
```

is an infinite loop.

For example, a Fortran **DO** loop could be written as

```
for (i = 1; i <= n; i = i + 1) ...
```

which is equivalent to

```
i = 1  
while (i <= n) {  
  ...  
  i = i + 1  
}
```

The initialization and increment of *i* have been moved into the **for** statement.

The **for** and **while** versions have the advantage that they will be done zero times if *n* is less than 1. This is not true of the **do**. In addition, the **break** and **next** statements work in a **for** loop.



The *increment* in a **for** need not be an arithmetic progression. The program

```
sum = 0.0
for (i = first; i > 0; i = ptr(i))
    sum = sum + value(i)
```

steps through a list (stored in an integer array *ptr*) until a zero pointer is found while adding up elements from a parallel array of values. Notice that the code also works correctly if the list is empty.

#### J. The "repeat-until" Statement

There are times when a test needs to be performed at the bottom of a loop after one pass through. This facility is provided by the **repeat-until** statement. The syntax for the **repeat-until** statement is

```
repeat
    ratfor statement
until (legal-Fortran-condition)
```

where *ratfor-statement* is done once, then the *condition* is evaluated. If it is true, the loop is exited; if it is false, another pass is made.

The **until** part is optional, so a **repeat** by itself is an infinite loop. A **repeat-until** loop can be exited by the use of a **stop**, **return**, or **break** statement or an implicit stop such as running out of input with a **READ** statement.

As stated before, a **break** statement causes an immediate exit from the enclosing **repeat-until** loop. A **next** statement will cause a skip to the bottom of a **repeat-until** loop (i.e., to the **until** part).

#### K. The "return" Statement

The standard Fortran mechanism for returning a value from a routine uses the name of the routine as a variable. This variable can be assigned a value. The last value stored in it is the value returned by the function. For example, in a Fortran routine named *equal*, the statements

```
equal = 0
return
```

cause *equal* to return zero.

The **ratfor** language provides a **return** statement similar to the C language **return** statement. In order to return a value from any routine, the **return** statement has the syntax

```
return ( expression )
```

where *expression* is the value to be returned.

If there is no parenthesized expression after **return**, no value is returned.

#### L. The "define" Statement

The **ratfor** language provides a **define** statement similar to the C language version. Any string of alphanumeric characters can be defined as a name. Whenever that name occurs in the input (delimited by nonalphanumerics), it is replaced by the rest of the definition line. (Comments and trailing white spaces are stripped off.) A defined name can be arbitrarily long and must begin with a letter.

Usually the **define** statement is used for symbolic parameters. The syntax of the **define** statement is

```
define name value
```



where *name* is a symbolic name that represents the quantity of *value*. For example:

```
define ROWS 100
define CLOS 50
dimension a(ROWS), b(ROWS, COLS)
      if (i > ROWS      i      j > COLS) ...
```

causes the preprocessor to replace the name *ROWS* with the value *100* and the name *COLS* with the value *50*. Alternately, definitions may be written as

```
define(ROWS, 100)
```

in which case the defining text is everything after the comma up to the right parenthesis. This allows multiple-line definitions.

#### M. The "include" Statement

The **ratfor** language provides an **include** statement similar to the **#include <...>** statement in C language. The syntax for this statement is

```
include file
```

which inserts the contents of the named file into the **ratfor** input file in place of the **include** statement. The standard usage is to place **COMMON** blocks on a file and use the **include** statement to include the common code whenever needed.

#### N. Free-Form Input

In **ratfor**, statements can be placed anywhere on a line. Long statements are continued automatically as are long conditions in **if**, **for**, and **until** statements. Blank lines are ignored. Multiple statements may appear on one line if they are separated by semicolons. No semicolon is needed at the end of a line if **ratfor** can make some reasonable guess about whether the statement ends there. Lines ending with any of the characters

```
= + - * , ! & ( _
```

are assumed to be continued on the next line. Underscores are discarded wherever they occur. All other characters remain as part of the statement.

Any statement that begins with an all-numeric field is assumed to be a Fortran label and placed in columns 1 through 5 upon output. Thus:

```
write(6, 100); 100 format("hello" )
```

is converted into

```
100      write(6, 100)
        format(5hello)
```

#### O. Translations

Text enclosed in matching single or double quotes is converted to *nH..* but is otherwise unaltered (except for formatting—it may get split across card boundaries during the reformatting process). Within quoted strings, the backslash (\) serves as an escape character; i.e., the next character is taken literally. This provides a way to get quotes and the backslash itself into quoted strings. For example:

```
" \\"
```



is a string containing a backslash and an apostrophe. (This is not the standard convention of doubled quotes, but it is easier to use and more general.)

Any line that begins with the character % is left absolutely unaltered except for stripping off the % and moving the line one position to the left. This is useful for inserting control cards and other things that should not be preprocessed (like an existing Fortran program). Use % only for ordinary statements not for the condition parts of **if**, **while**, etc., or the output may come out in an unexpected place.

The following character translations are made (except within single or double quotes or on a line beginning with a %):

|    |       |
|----|-------|
| == | .eq.  |
| != | .ne.  |
| >  | .gt.  |
| >= | .ge.  |
| <  | .lt.  |
| <= | .le.  |
| &  | .and. |
|    | .or.  |
| !  | .not. |
| ¬  | .not. |

In addition, the following translations are provided for input devices with restricted character sets:

|     |   |
|-----|---|
| [   | { |
| ]   | } |
| \$( | { |
| \$) | } |

#### P. Warnings

The **ratfor** preprocessor catches certain syntax errors (such as missing braces), **else** statements without **if** statements, and most errors involving missing parentheses in statements.

All other errors are reported by the Fortran compiler. Unfortunately, the Fortran compiler prints messages in terms of generated Fortran code and not in terms of the **ratfor** code. This makes it difficult to locate **ratfor** statements that contain errors.

The keywords are reserved. Using **if**, **else**, **while**, etc., as variable names will cause considerable problems. Likewise, spaces within keywords and use of the Arithmetic IF will cause problems.

The Fortran **nH** convention is not recognized by **ratfor**. Use quotes instead.



## NOTES



# Support Tools Guide

## UNIX System



**Trademarks:**

|                    |                           |
|--------------------|---------------------------|
| MUNIX, CADMUS      | for PCS                   |
| UNIX               | for Bell Laboratories     |
| DEC, PDP, VAX      | for DEC                   |
| MASSBUS, UNIBUS    |                           |
| KODAK, EKTAMATIC   | for Eastman Kodak Company |
| Mohrflow, Mohrdry, | for Mohr Lino-Saw Comp.   |
| Mohrchem           |                           |
| TEKTRONIX          | for Tektronik, Inc.       |
| TELETYPE           | for Teletype Corporation  |
| TRENDATA 4000A®    | for Trendata Corporation  |
| Versatec           | for Versatec Corporation  |
| DIABLO             | for Xerox Corporation     |

Copyright 1984 by  
PCS GmbH, Pfälzer-Wald-Strasse 38, D-8000 München 90, tel. (089) 87804-0

The information contained herein is the property of PCS and shall neither be reproduced in whole or in part without PCS's prior written approval nor be implied to grant any license to make, use or sell equipment manufactured herewith.

PCS reserves the right to make changes without notice in the specifications and materials contained herein and shall not be responsible for any damages (including consequential) caused by reliance on the materials presented.



## SUPPORT TOOLS GUIDE

## UNIX SYSTEM

| CONTENTS                                                        | PAGE |
|-----------------------------------------------------------------|------|
| 1. INTRODUCTION . . . . .                                       | 9    |
| 2. A PROGRAM FOR MAINTAINING COMPUTER PROGRAMS (make) . . . . . | 11   |
| GENERAL . . . . .                                               | 11   |
| BASIC FEATURES . . . . .                                        | 13   |
| DESCRIPTION FILES AND SUBSTITUTIONS . . . . .                   | 15   |
| COMMAND USAGE . . . . .                                         | 16   |
| SUFFIXES AND TRANSFORMATION RULES . . . . .                     | 17   |
| IMPLICIT RULES . . . . .                                        | 18   |
| SUGGESTIONS AND WARNINGS . . . . .                              | 19   |
| 3. AUGMENTED VERSION OF MAKE . . . . .                          | 21   |
| GENERAL . . . . .                                               | 21   |
| THE ENVIRONMENT VARIABLES . . . . .                             | 21   |
| RECURSIVE MAKEFILES . . . . .                                   | 22   |
| FORMAT OF SHELL COMMANDS WITHIN make . . . . .                  | 23   |
| ARCHIVE LIBRARIES . . . . .                                     | 23   |
| SOURCE CODE CONTROL SYSTEM FILE NAMES: THE TILDE . . . . .      | 24   |
| THE NULL SUFFIX . . . . .                                       | 25   |
| INCLUDE FILES . . . . .                                         | 26   |
| INVISIBLE SCCS MAKEFILES . . . . .                              | 26   |
| DYNAMIC DEPENDENCY PARAMETERS . . . . .                         | 26   |



| CONTENTS                                             | PAGE |
|------------------------------------------------------|------|
| EXTENSIONS OF \$*, \$@, AND \$< . . . . .            | 27   |
| OUTPUT TRANSLATIONS . . . . .                        | 27   |
| 4. SOURCE CODE CONTROL SYSTEM USER'S GUIDE . . . . . | 41   |
| GENERAL . . . . .                                    | 41   |
| SCCS FOR BEGINNERS . . . . .                         | 41   |
| A. Terminology . . . . .                             | 42   |
| B. Creating an SCCS File via "admin" . . . . .       | 42   |
| C. Retrieving a File via "get" . . . . .             | 42   |
| D. Recording Changes via "delta" . . . . .           | 43   |
| E. Additional Information About "get" . . . . .      | 44   |
| F. The "help" Command . . . . .                      | 45   |
| DELTA NUMBERING . . . . .                            | 45   |
| SCCS COMMAND CONVENTIONS . . . . .                   | 47   |
| SCCS COMMANDS . . . . .                              | 48   |
| A. The "get" Command . . . . .                       | 49   |
| B. The "delta" Command . . . . .                     | 55   |
| C. The "admin" Command . . . . .                     | 57   |
| D. The "prs" Command . . . . .                       | 59   |
| E. The "help" Command . . . . .                      | 61   |
| F. The "rmDEL" Command . . . . .                     | 61   |
| G. The "cdc" Command . . . . .                       | 62   |
| H. The "what" Command . . . . .                      | 62   |
| I. The "scsdiff" Command . . . . .                   | 63   |
| J. The "comb" Command . . . . .                      | 63   |
| K. The "val" Command . . . . .                       | 64   |
| SCCS FILES . . . . .                                 | 64   |



|    | CONTENTS                                 | PAGE |
|----|------------------------------------------|------|
| A. | Protection . . . . .                     | 64   |
| B. | Formatting . . . . .                     | 65   |
| C. | Auditing . . . . .                       | 65   |
|    | AN SCCS INTERFACE PROGRAM . . . . .      | 66   |
| A. | General . . . . .                        | 66   |
| B. | Function . . . . .                       | 66   |
| C. | A Basic Program . . . . .                | 67   |
| D. | Linking and Use . . . . .                | 67   |
| 5. | THE M4 MACRO PROCESSOR . . . . .         | 71   |
|    | GENERAL . . . . .                        | 71   |
|    | DEFINING MACROS . . . . .                | 71   |
|    | ARGUMENTS . . . . .                      | 74   |
|    | ARITHMETIC BUILT-INS . . . . .           | 74   |
|    | FILE MANIPULATION . . . . .              | 75   |
|    | SYSTEM COMMAND . . . . .                 | 75   |
|    | CONDITIONALS . . . . .                   | 76   |
|    | STRING MANIPULATION . . . . .            | 76   |
|    | PRINTING . . . . .                       | 77   |
| 6. | THE "awk" PROGRAMMING LANGUAGE . . . . . | 81   |
|    | GENERAL . . . . .                        | 81   |
| A. | Usage . . . . .                          | 81   |
| B. | Program Structure . . . . .              | 81   |
| C. | Records and Fields . . . . .             | 82   |
| D. | Printing . . . . .                       | 82   |
|    | PATTERNS . . . . .                       | 83   |
| A. | "BEGIN" and "END" . . . . .              | 83   |



| CONTENTS                                                       | PAGE |
|----------------------------------------------------------------|------|
| B. Regular Expressions . . . . .                               | 84   |
| C. Relational Expressions . . . . .                            | 85   |
| D. Combinations of Patterns . . . . .                          | 85   |
| E. Pattern Ranges . . . . .                                    | 85   |
| ACTIONS . . . . .                                              | 86   |
| A. Built-in Functions . . . . .                                | 86   |
| B. Variables, Expressions, and Assignments . . . . .           | 86   |
| C. Field Variables . . . . .                                   | 87   |
| D. String Concatenation . . . . .                              | 88   |
| E. Arrays . . . . .                                            | 88   |
| F. Flow-of-Control Statements . . . . .                        | 89   |
| 7. ARBITRARY PRECISION DESK CALCULATOR LANGUAGE (BC) . . . . . | 91   |
| GENERAL . . . . .                                              | 91   |
| SIMPLE COMPUTATIONS WITH INTERGERS . . . . .                   | 91   |
| BASES . . . . .                                                | 92   |
| SCALING . . . . .                                              | 93   |
| FUNCTIONS . . . . .                                            | 94   |
| SUBSCRIPTED VARIABLES . . . . .                                | 95   |
| CONTROL STATEMENTS . . . . .                                   | 95   |
| ADDITIONAL FEATURES . . . . .                                  | 97   |
| 8. INTERACTIVE DESK CALCULATOR (DC) . . . . .                  | 105  |
| GENERAL . . . . .                                              | 105  |
| DC COMMANDS . . . . .                                          | 105  |
| INTERNAL REPRESENTATION OF NUMBERS . . . . .                   | 107  |
| THE ALLOCATOR . . . . .                                        | 107  |
| INTERNAL ARITHMETIC . . . . .                                  | 108  |



| CONTENTS                                      | PAGE |
|-----------------------------------------------|------|
| ADDITION AND SUBTRACTION . . . . .            | 108  |
| MULTIPLICATION . . . . .                      | 108  |
| DIVISION . . . . .                            | 108  |
| REMAINDER . . . . .                           | 109  |
| SQUARE ROOT . . . . .                         | 109  |
| EXPONENTIATION . . . . .                      | 109  |
| INPUT CONVERSION AND BASE . . . . .           | 109  |
| OUTPUT COMMANDS . . . . .                     | 109  |
| OUTPUT FORMAT AND BASE . . . . .              | 110  |
| INTERNAL REGISTERS . . . . .                  | 110  |
| STACK COMMANDS . . . . .                      | 110  |
| SUBROUTINE DEFINITIONS AND CALLS . . . . .    | 110  |
| INTERNAL REGISTERS—PROGRAMMING DC . . . . .   | 110  |
| PUSHDOWN REGISTERS AND ARRAYS . . . . .       | 110  |
| MISCELLANEOUS COMMANDS . . . . .              | 110  |
| DESIGN CHOICES . . . . .                      | 111  |
| 9. LEXICAL ANALYZER GENERATOR (LEX) . . . . . | 113  |
| GENERAL . . . . .                             | 113  |
| LEX SOURCE . . . . .                          | 115  |
| LEX REGULAR EXPRESSIONS . . . . .             | 115  |
| A. Operators . . . . .                        | 116  |
| B. Character Classes . . . . .                | 116  |
| C. Arbitrary Character . . . . .              | 117  |
| D. Optional Expressions . . . . .             | 117  |
| E. Repeated Expressions . . . . .             | 117  |
| F. Alternation and Grouping . . . . .         | 117  |



| CONTENTS                                           | PAGE |
|----------------------------------------------------|------|
| G. Context Sensitivity . . . . .                   | 118  |
| H. Repetitions and Definitions . . . . .           | 118  |
| LEX ACTIONS . . . . .                              | 119  |
| AMBIGUOUS SOURCE RULES . . . . .                   | 121  |
| LEX SOURCE DEFINITIONS . . . . .                   | 123  |
| USAGE . . . . .                                    | 124  |
| LEX AND YACC . . . . .                             | 124  |
| EXAMPLES . . . . .                                 | 125  |
| LEFT CONTEXT SENSITIVITY . . . . .                 | 126  |
| CHARACTER SET . . . . .                            | 128  |
| SUMMARY OF SOURCE FORMAT . . . . .                 | 128  |
| CAVEATS AND BUGS . . . . .                         | 129  |
| 10. YET ANOTHER COMPLIER—COMPLIER (yacc) . . . . . | 131  |
| GENERAL . . . . .                                  | 131  |
| BASIC SPECIFICATIONS . . . . .                     | 133  |
| ACTIONS . . . . .                                  | 134  |
| LEXICAL ANALYSIS . . . . .                         | 137  |
| PARSER OPERATION . . . . .                         | 138  |
| AMBIGUITY AND CONFLICTS . . . . .                  | 141  |
| PRECEDENCE . . . . .                               | 145  |
| ERROR HANDLING . . . . .                           | 147  |
| THE "yacc" ENVIRONMENT . . . . .                   | 149  |
| HINTS FOR PREPARING SPECIFICATIONS . . . . .       | 150  |
| A. Input Style . . . . .                           | 150  |
| B. Left Recursion . . . . .                        | 150  |
| C. Lexical Tie-ins . . . . .                       | 151  |



| CONTENTS                                            | PAGE |
|-----------------------------------------------------|------|
| D. Reserved Words . . . . .                         | 152  |
| ADVANCED TOPICS . . . . .                           | 152  |
| A. Simulating Error and Accept in Actions . . . . . | 152  |
| B. Accessing Values in Enclosing Rules . . . . .    | 152  |
| C. Support for Arbitrary Value Types . . . . .      | 153  |



SUPPORT TOOLS

ISSUE 1

6/82

NOTES



## 1. INTRODUCTION

The SUPPORT TOOLS volume is a description of the various software "tools" which may aid the UNIX operating system user. The following paragraphs contain a brief description of each section.

The section A PROGRAM FOR MAINTAINING COMPUTER PROGRAMS (`make`) describes a software tool for maintaining, updating, and regenerating groups of computer programs. The many activities of program development and maintenance are made simpler by the `make` program.

The section AUGMENTED VERSION OF "make" describes the modifications made to handle many of the problems within the original `make` program.

The section SOURCE CODE CONTROL SYSTEM USER'S GUIDE describes the collection of SCCS programs under the UNIX operating system. The SCCS programs act as a "custodian" over the UNIX system files.

The section THE M4 MACRO PROCESSOR describes the front end for rational Fortran and C programming language.

The section THE "awk" PROGRAMMING LANGUAGE describes a software tool designed to make many common information retrieval and text manipulation tasks easy to state and to perform.

The section ARBITRARY PRECISION DESK CALCULATOR LANGUAGE (BC) describes a compiler for doing arbitrary precision arithmetic on the UNIX operating system.

The section INTERACTIVE DESK CALCULATOR (DC) describes a program implemented on the UNIX operating system to do arbitrary-precision integer arithmetic.

The section LEXICAL ANALYZER GENERATOR (Lex) describes a software tool designed for lexical processing of character input streams.

The section YET ANOTHER COMPILER — COMPILER (`yacc`) describes the `yacc` program. The `yacc` program provides a general tool for imposing structure on the input to a computer program.

The support tools provide an added dimension to the basic UNIX software commands. The "tools" described will enable the user to fully utilize the UNIX operating system.



## NOTES



## 2. A PROGRAM FOR MAINTAINING COMPUTER PROGRAMS (*make*)

### GENERAL

In a programming project, a common practice is to divide large programs into smaller pieces that are more manageable. The pieces may require several different treatments such as being processed by a macro processor or sophisticated program generators (e.g., *Yacc* or *Lex*). The project continues to become more complex as the output of these generators may be compiled with special options and with certain definitions and declarations. A sequence of code transformations develops which is difficult to remember. The resulting code may need further transformation by loading the code with certain libraries under control of special options. Related maintenance activities also complicate the process further by running test scripts and installing validated modules. Another activity which complicates program development is a long editing session. A programmer may lose track of the files changed and the object modules still valid especially when a change to a declaration can make a dozen other files obsolete. The programmer must also remember to compile a routine that has been changed or that uses changed declarations.

A programmer can easily forget which files depend on which other files, which files have been modified recently, which files need to be reprocessed or recompiled after a change in some part of the source, and the exact sequence of operations needed to make or exercise a new version of the program. The many activities of program development and maintenance are made simpler by the *make* program. The *make* program is used to maintain, update, and regenerate groups of computer programs.

The *make* program provides a method for maintaining up-to-date versions of programs that result from many operations on a number of files. The *make* program can keep track of the sequence of commands that create certain files and the list of files that require other files to be current before the operations can be done. Whenever a change is made in any part of a program, the *make* command will create the proper files simply, correctly, and with a minimum amount of effort. The *make* program also provides a simple macro substitution facility and the ability to encapsulate commands in a single file for convenient administration.

The basic operation of *make* is to find the name of a needed target file in the description, ensure that all of the files on which it depends exist and are up to date, and then create the target file if it has not been modified since its generators were. The descriptor file really defines the graph of dependencies. The *make* program determines the necessary work by performing a depth-first search of the graph of dependencies.

If the information on interfile dependences and command sequences is stored in a file, the simple command

```
make
```

is frequently sufficient to update the interesting files regardless of the number edited since the last *make*. In most cases, the description file is easy to write and changes infrequently. It is usually easier to type the *make* command than to issue even one of the needed operations, so the typical cycle of program development operations becomes:

```
think - edit - make - test ...
```

The *make* program is most useful for medium-sized programming projects. The *make* program does not solve the problems of maintaining multiple source versions or of describing huge programs.

As an example of the use of *make*, the description file used to maintain the *make* command is given. The code for *make* is spread over a number of C language source files and a *Yacc* grammar. The description file contains:

```
# Description file for the Make command

P = und -3 | opr -r2 # send to be printed
FILES = Makefile version.c defs main.c doname.c misc.c files.c dosys.c
        gram.y lex.c gcos.c
```



```

OBJECTS = version.o main.o doname.o misc.o files.o dosys.o gram.o
LIBES= -ls
LINT = lint -p
CFLAGS = -O

```

```

make: $(OBJECTS)
      cc $(CFLAGS) $(OBJECTS) $(LIBES) -o make
      size make

```

```

$(OBJECTS): defs
gram.o: lex.c

```

```

cleanup:
      -rm *.o gram.c
      -du

```

```

install:
      @size make /usr/bin/make
      cp make /usr/bin/make ; rm make

```

```

print: $(FILES)      # print recently changed files
      pr $? ! $P
      touch print

```

```

test:
      make -dp ! grep -v TIME >1zap
      /usr/bin/make -dp ! grep -v TIME >2zap
      diff 1zap 2zap
      rm 1zap 2zap

```

```

lint: dosys.c doname.c files.c main.c misc.c version.c gram.c
      $(LINT) dosys.c doname.c files.c main.c misc.c version.c gram.c
      rm gram.c

```

```

arch:
      ar uv /sys/source/s2/make.a $(FILES)

```

The **make** program usually prints out each command before issuing it.

The following output results from typing the simple command **make** in a directory containing only the source and description file:

```

cc -O -c version.c
cc -O -c main.c
cc -O -c doname.c
cc -O -c misc.c
cc -O -c files.c
cc -O -c dosys.c
yacc gram.y
mv y.tab.c gram.c
cc -O -c gram.c
cc version.o main.o doname.o misc.o files.o dosys.o
      gram.o -ls -o make
13188+3348+3044 = 19580b = 046174b

```



Although none of the source files or grammars were mentioned by name in the description file, **make** found them using its suffix rules and issued the needed commands. The string of digits results from the **size make** command. The printing of the command line itself was suppressed by an @ sign. The @ sign on the **size** command in the description file suppressed the printing of the command, so only the sizes are written.

The last few entries in the description file are useful maintenance sequences. The "print" entry prints only the files changed since the last **make print** command. A zero-length file *print* is maintained to keep track of the time of the printing. The \$? macro in the command line then picks up only the names of the files changed since *print* was touched. The printed output can be sent to a different printer or to a file by changing the definition of the P macro as follows:

```
make print " P = opr -sp "
           or
make print " P= cat >zap "
```

## BASIC FEATURES

The basic operation of **make** is to update a target file by ensuring that all of the files on which the target file depends exist and are up-to-date. The target file is created if it has not been modified since the dependents were. The **make** program does a depth-first search of the graph of dependences. The operation of the command depends on the ability to find the date and time that a file was last modified.

To illustrate, consider a simple example in which a program named **prog** is made by compiling and loading three C language files *x.c*, *y.c*, and *z.c* with the **ls** library. By convention, the output of the C language compilations will be found in files named *x.o*, *y.o*, and *z.o*. Assume that the files *x.c* and *y.c* share some declarations in a file named *defs*, but that *z.c* does not. That is, *x.c* and *y.c* have the line:

```
#include "defs"
```

The following text describes the relationships and operations:

```
prog: x.o y.o z.o
    cc x.o y.o z.o -ls -o prog

x.o y.o: defs
```

If this information were stored in a file named *makefile*, the command

```
make
```

would perform the operations needed to recreate **prog** after any changes had been made to any of the four source files *x.c*, *y.c*, *z.c*, or *defs*.

The **make** program operates using the following three sources of information:

- a user-supplied description file
- file names and "last-modified" times from the file system
- built-in rules to bridge some of the gaps.

In our example, the first line states that **prog** depends on three ".o" files. Once these object files are current, the second line describes how to load them to create **prog**. The third line states that *x.o* and *y.o* depend on the



file *defs*. From the file system, **make** discovers that there are three ".c" files corresponding to the needed ".o" files and uses built-in information on how to generate an object from a source file (i.e., issue a "cc -c" command).

Not taking advantage of **make**'s innate knowledge results in the following longer description file using the same example:

```
prog : x.o y.o z.o
      cc x.o y.o z.o -ls -o prog
x.o : x.c defs
      cc -c x.c
y.o : y.c defs
      cc -c y.c
z.o : z.c
      cc -c z.c
```

If none of the source or object files had changed since the last time **prog** was made, all of the files would be current, and the command

```
make
```

would just announce this fact and stop. If, however, the *defs* file had been edited, *x.c* and *y.c* (but not *z.c*) would be recompiled; and then **prog** would be created from the new ".o" files. If only the file *y.c* had changed, only it would be recompiled; but it would still be necessary to reload **prog**. If no target name is given on the **make** command line, the first target mentioned in the description is created; otherwise, the specified targets are made. The command

```
make x.o
```

would recompile *x.o* if *x.c* or *defs* had changed.

If the file exists after the commands are executed, the file's time of last modification is used in further decisions. If the file does not exist after the commands are executed, the current time is used in making further decisions. Often a method useful to programmers is to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of **make**'s ability to generate files and substitute macros. Thus, an entry "save" might be included to copy a certain set of files, or an entry "clean-up" might be used to throw away unneeded intermediate files. In other cases, one may maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

The **make** program has a simple macro mechanism for substituting in dependency lines and command strings. Macros are defined by command arguments or description file lines with embedded equal signs. A macro is invoked by preceding the name by a dollar sign. Macro names longer than one character must be parenthesized. The name of the macro is either the single character after the dollar sign or a name inside parentheses. The following are valid macro invocations:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two invocations are identical. A \$\$ is a dollar sign.

The \$\*, \$@, \$?, and \$< are four special macros which change values during the execution of the command. (These four macros are described in the part "DESCRIPTION FILES AND SUBSTITUTIONS".) The following



fragment shows assignment and use of some macros:

```
OBJECTS = x.o y.o z.o
LIBES = -lS
prog: $(OBJECTS)
      cc $(OBJECTS) $(LIBES) -o prog
```

...

The **make** command loads the three object files with the **lS** library. The command

```
make "LIBES= -ll -lS "
```

loads them with both the Lex (**-ll**) and the standard (**-lS**) libraries since macro definitions on the command line override definitions in the description. Remember to quote arguments with embedded blanks in UNIX software commands.

### DESCRIPTION FILES AND SUBSTITUTIONS

A description file contains the following information:

- macro definitions
- dependency information
- executable commands.

The comment convention is that a sharp (#) and all characters on the same line after a sharp are ignored. Blank lines and lines beginning with a sharp (#) are totally ignored. If a noncomment line is too long, the line can be continued by using a backslash. If the last character of a line is a backslash, then the backslash, the new line, and all following blanks and tabs are replaced by a single blank.

A macro definition is a line containing an equal sign not preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped). The following are valid macro definitions:

```
2 = xyz
abc = -ll -ly -lS
LIBES =
```

The last definition assigns **LIBES** the null string. A macro that is never explicitly defined has the null string as the macro's value.

Macro definitions may also appear on the **make** command line while other lines give information about target files. The general form of an entry is:

```
target1 [target2 ..] [:] [dependent1 ..] [; commands] [# ..]
[(tab) commands] [# ..]
...
```

Items inside brackets may be omitted. Targets and dependents are strings of letters, digits, periods, and slashes. Shell metacharacters such as "\*" and "?" are expanded. Commands may appear either after a semicolon on a dependency line or on lines beginning with a tab immediately following a dependency line. A command is any string of characters not including a sharp (#) except when the sharp is in quotes or not including a new line.



A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type. For the usual single-colon case, a command sequence may be associated with at most one dependency line. If the target is out of date with any of the dependents on any of the lines and a command sequence is specified (even a null one following a semicolon or tab); it is executed; otherwise, a default creation rule may be invoked. In the double-colon case, a command sequence may be associated with each dependency line; if the target is out of date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. This detailed form is of particular value in updating archive-type files.

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the shell after substituting for macros. The printing is suppressed in the silent mode or if the command line begins with an @ sign. **Make** normally stops if any command signals an error by returning a nonzero error code. Errors are ignored if the **-i** flag has been specified on the **make** command line, if the fake target name ".IGNORE" appears in the description file, or if the command string in the description file begins with a hyphen. Some UNIX software commands return meaningless status. Because each command line is passed to a separate invocation of the shell, care must be taken with certain commands (e.g., **cd** and shell control commands) that have meaning only within a single shell process. These results are forgotten before the next line is executed.

Before issuing any command, certain internally maintained macros are set. The **\$@** macro is set to the full target name of the current target. The **\$@** macro is evaluated only for explicitly named dependencies. The **\$?** macro is set to the string of names that were found to be younger than the target. The **\$?** macro is evaluated when explicit rules from the *makefile* are evaluated. If the command was generated by an implicit rule, the **\$<** macro is the name of the related file that caused the action; and the **\$\*** macro is the prefix shared by the current and the dependent file names. If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name ".DEFAULT" are used. If there is no such name, **make** prints a message and stops.

## COMMAND USAGE

The **make** command takes macro definitions, flags, description file names, and target file names as arguments in the form:

```
make [ flags ] [ macro definitions ] [ targets ]
```

The following summary of command operations explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments made. Command-line macros override corresponding definitions found in the description files. Next, the flag arguments are examined. The permissible flags are:

- i** Ignore error codes returned by invoked commands. This mode is entered if the fake target name ".IGNORE" appears in the description file.
- s** Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name ".SILENT" appears in the description file.
- r** Do not use the built-in rules.
- n** No execute mode. Print commands, but do not execute them. Even lines beginning with an "@" sign are printed.
- t** Touch the target files (causing them to be up to date) rather than issue the usual commands.
- q** Question. The **make** command returns a zero or nonzero status code depending on whether the target file is or is not up to date.



- p Print out the complete set of macro definitions and target descriptions.
- d Debug mode. Print out detailed information on files and times examined.
- f Description file name. The next argument is assumed to be the name of a description file. A file name of "-" denotes the standard input. If there are no "-f" arguments, the file named *makefile* or *Makefile* in the current directory is read. The contents of the description files override the built-in rules if they are present.

Finally, the remaining arguments are assumed to be the names of targets to be made, and the arguments are done in left to right order. If there are no such arguments, the first name in the description files that does not begin with a period is "made".

### SUFFIXES AND TRANSFORMATION RULES

The **make** program does not know what file name suffixes are interesting or how to transform a file with one suffix into a file with another suffix. This information is stored in an internal table that has the form of a description file. If the **-r** flag is used, the internal table is not used.

The list of suffixes is actually the dependency list for the name ".SUFFIXES". The **make** program searches for a file with any of the suffixes on the list. If such a file exists and if there is a transformation rule for that combination, **make** transforms a file with one suffix into a file with another suffix. The transformation rule names are the concatenation of the two suffixes. The name of the rule to transform a *.r* file to a *.o* file is thus *.r.o*. If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule *.r.o* is used. If a command is generated by using one of these suffixing rules, the macro **\$\*** is given the value of the stem (everything but the suffix) of the name of the file to be made; and the macro **\$<** is the name of the dependent that caused the action.

The order of the suffix list is significant since the list is scanned from left to right. The first name formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can add an entry for ".SUFFIXES" in his own description file. The dependents will be added to the usual list. A ".SUFFIXES" line without any dependents deletes the current list. It is necessary to clear the current list if the order of names is to be changed. The following is an excerpt from the default rules file:

```
.SUFFIXES : .o .c .e .r .f .y .yr .ye .l .s
YACC = yacc
YACCR = yacc -r
YACCE = yacc -e
YFLAGS =
LEX = lex
LFLAGS =
CC = cc
AS = as -
CFLAGS =
RC = ec
RFLAGS =
EC = ec
EFLAGS =
FFlags =
.c.o :
    $(CC) $(CFLAGS) -c $<
.e.o .r.o .f.o :
    $(EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<
```



```
.s.o :
$(AS) -o $@ $<
.y.o :
$(YACC) $(YFLAGS) $<
$(CC) $(CFLAGS) -c y.tab.c
rm y.tab.c
mv y.tab.o $@
.y.c :
$(YACC) $(YFLAGS) $<
mv y.tab.c $@
```

### IMPLICIT RULES

The **make** program uses a table of interesting suffixes and a set of transformation rules to supply default dependency information and implied commands. The default suffix list is:

|            |                            |
|------------|----------------------------|
| <b>.o</b>  | Object file                |
| <b>.c</b>  | C source file              |
| <b>.e</b>  | Efl source file            |
| <b>.r</b>  | Ratfor source file         |
| <b>.f</b>  | Fortran source file        |
| <b>.s</b>  | Assembler source file      |
| <b>.y</b>  | Yacc-C source grammar      |
| <b>.yr</b> | Yacc-Ratfor source grammar |
| <b>.ye</b> | Yacc-Efl source grammar    |
| <b>.l</b>  | Lex source grammar         |

Figure 2.1 summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.

If the file **x.o** were needed and there were an **x.c** in the description or directory, the **x.o** file would be compiled. If there were also an **x.l**, that grammar would be run through Lex before compiling the result. However, if there were no **x.c** but there were an **x.l**, **make** would discard the intermediate C language file and use the direct link as shown in Fig. 2.1.

It is possible to change the names of some of the compilers used in the default or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros **AS**, **CC**, **RC**, **EC**, **YACC**, **YACCR**, **YACCE**, and **LEX**. The command

```
make CC=newcc
```

will cause the **newcc** command to be used instead of the usual C language compiler. The macros **CFLAGS**, **RFLAGS**, **EFLAGS**, **YFLAGS**, and **LFLAGS** may be set to cause these commands to be issued with optional flags. Thus

```
make "CFLAGS=-O"
```

causes the optimizing C language compiler to be used.



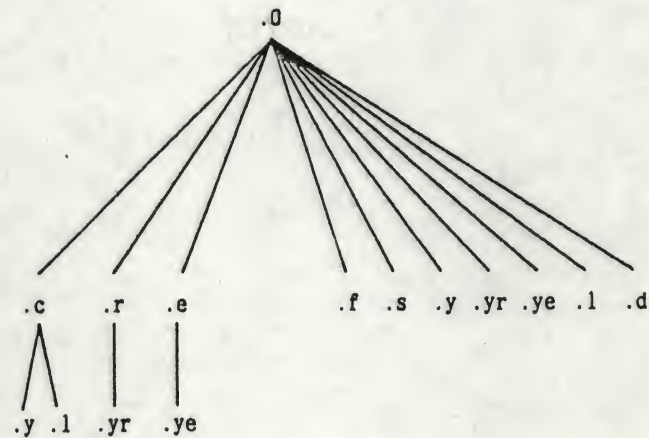


Fig. 2.1 — Summary of Default Transformation Path

### SUGGESTIONS AND WARNINGS

The most common difficulties arise from **make**'s specific meaning of dependency. If file **x.c** has a "#include "defs" " line, then the object file **x.o** depends on **defs**; the source file **x.c** does not. If **defs** is changed, nothing is done to the file **x.c** while file **x.o** must be recreated.

To discover what **make** would do, the **-n** option is very useful. The command

```
make -n
```

orders **make** to print out the commands which **make** would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be mild in character (e.g., adding a new definition to an include file), the **-t** (touch) option can save a lot of time. Instead of issuing a large number of superfluous recompilations, **make** updates the modification times on the affected file. Thus, the command

```
make -ts
```

"touch silently" causes the relevant files to appear up to date. Obvious care is necessary since this mode of operation subverts the intention of **make** and destroys all memory of the previous relationships.

The debugging flag (**-d**) causes **make** to print out a very detailed description of what it is doing including the file times. The output is verbose and recommended only as a last resort.



## NOTES



### 3. AUGMENTED VERSION OF MAKE

#### GENERAL

This section describes an augmented version of the **make** command of the UNIX operating system. The augmented version is upward compatible with the old version. This section describes and gives examples of only the additional features. Further possible developments for **make** are also discussed. Some justification will be given for the chosen implementation, and examples will demonstrate the additional features.

The **make** command was perceived as an excellent program administrative tool and has been used extensively in at least one project for over 2 years. However, **make** had the following shortcomings:

- handling of libraries was tedious
- handling of the Source Code Control System (SCCS) file name format was difficult or impossible
- environment variables were completely ignored by **make**
- the general lack of ability to maintain files in a remote directory.

These shortcomings hindered large scale use of **make** as a program support tool.

The **make** program has been modified to handle the problems above. The additional features are within the original syntactic framework of **make** and few if any new syntactical entities have been introduced. A notable exception is the *include* file capability. Further, most of the additions result in a "Don't know how to make ..." message from the old version of **make**.

The following paragraphs describe with examples the additional features of the **make** program. In general, the examples are taken from existing *makefiles*. Also, the tables are examples of working *makefiles*.

#### THE ENVIRONMENT VARIABLES

Environment variables are read and added to the macro definitions each time **make** executes. Precedence is a prime consideration in doing this properly. The following describes **make's** interaction with the environment. A new macro, **MAKEFLAGS**, is maintained by **make**. The new macro is defined as the collection of all input flag arguments into a string (without minus signs). The new macro is exported and thus accessible to further invocations of **make**. Command line flags and assignments in the *makefile* update **MAKEFLAGS**. Thus, to describe how the environment interacts with **make**, the **MAKEFLAGS** macro (environment variable) must be considered.

When executed, **make** assigns macro definitions in the following order:

1. Read the **MAKEFLAGS** environment variable. If it is not present or null, the internal **make** variable **MAKEFLAGS** is set to the null string. Otherwise, each letter in **MAKEFLAGS** is assumed to be an input flag argument and is processed as such. (The only exceptions are the **-f**, **-p**, and **-r** flags.)
2. Read and set the input flags from the command line. The command line adds to the previous settings from the **MAKEFLAGS** environment variable.
3. Read macro definitions from the command line. These are made *not resettable*. Thus, any further assignments to these names are ignored.
4. Read the internal list of macro definitions. These are found in the file *rules.c* of the source for **make**. (See Table 3.A for the complete *makefile* which represents the internally defined macros and rules of the



current version of **make**. Thus, if **make -r ...** is typed and a *makefile* includes the *makefile* in Table 3.A, the results would be identical to excluding the **-r** option and the *include* line in the *makefile*. The Table 3.A output can be reproduced by the following:

```
make -fp - < /dev/null 2>/dev/null
```

The output will appear on the standard output.) They give default definitions for the C language compiler (CC=cc), the assembler (AS=as), etc.

5. Read the environment. The environment variables are treated as macro definitions and marked as *exported* (in the shell sense). (**Note:** MAKEFLAGS will be read and set again.) However, since MAKEFLAGS is not an internally defined variable (in *rules.c*), this has the effect of doing the same assignment twice. The exception to this is when MAKEFLAGS is assigned on the command line. (The reason it was read previously was to turn the debug flag on before anything else was done.)
6. Read the *makefile(s)*. The assignments in the *makefile(s)* will override the environment. This order was chosen so when a *makefile* is read and executed the user knows what to expect. That is, the user gets what is seen unless the **-e** flag is used. The **-e** is an additional command line flag which tells **make** to have the environment override the *makefile* assignments. Thus, if **make -e ...** is typed, the variables in the environment override the definitions in the *makefile*. (**Note:** There is no way to override the command line assignments.) Also note that MAKEFLAGS will override the environment if assigned. (This would be useful for further invocations of **make** from the current *makefile*.)

It may be clearer to list the precedence of assignments. Thus, in order from least binding to most binding, the precedence of assignments is as follows:

1. internal definitions (from *rules.c*)
2. environment
3. *makefile(s)*
4. command line.

The **-e** flag has the effect of changing the order to:

1. internal definitions (from *rules.c*)
2. *makefile(s)*
3. environment
4. command line.

This order is general enough to allow a programmer to define a *makefile* or set of *makefiles* whose parameters are dynamically definable.

## RECURSIVE MAKEFILES

Another feature was added to **make** concerning the environment and recursive invocations. If the sequence **\$(MAKE)** appears anywhere in a shell command line, the line will be executed even if the **-n** flag is set. Since the **-n** flag is exported across invocations of **make** (through the MAKEFLAGS variable), the only thing which will actually get executed is the **make** command itself. This feature is useful when a hierarchy of



*makefile(s)* describes a set of software subsystems. For testing purposes, **make -n ...** can be executed and everything that would have been done will get printed out including output from lower level invocations of **make**.

#### FORMAT OF SHELL COMMANDS WITHIN **make**

The **make** program remembers embedded new lines and tabs in shell command sequences. Thus, if the programmer puts a *for* loop in the makefile with indentation, when **make** prints it out, it retains the indentation and backslashes. The output can still be piped to the shell and is readable. This is obviously a cosmetic change; no new function is gained.

#### ARCHIVE LIBRARIES

The **make** program has an improved interface to archive libraries. Due to a lack of documentation, most people are probably not aware of the current syntax of addressing members of archive libraries. The previous version of **make** allows a user to name a member of a library in the following manner:

```
lib(object.o)
or
lib((_localtime))
```

where the second method actually refers to an entry point of an object file within the library. (Make looks through the library, locates the entry point, and translates it to the correct object file name.)

To use this procedure to maintain an archive library, the following type of *makefile* is required:

```
lib: lib(ctime.o)
    $(CC) -c -O ctime.c
    ar rv lib ctime.o
    rm ctime.o
lib: lib(fopen.o)
    $(CC) -c -O fopen.c
    ar rv lib fopen.o
    rm fopen.o
...and so on for each object ...
```

This is tedious and error prone. Obviously, the command sequences for adding a C language file to a library are the same for each invocation; the file name being the only difference each time. (This is true in most cases.)

The current version gives the user access to a rule for building libraries. The handle for the rule is the ".a" suffix. Thus, a ".c.a" rule is the rule for compiling a C language source file, adding it to the library, and removing the ".o" cadaver. Similarly, the ".y.a", the ".s.a", and the ".l.a" rules rebuild YACC, assembler, and LEX files, respectively. The current archive rules defined internally are ".c.a", ".c~.a", and ".s~.a". (The tilde (~) syntax will be described shortly.) The user may define in makefile other rules needed.

The above 2-member library is then maintained with the following shorter makefile:

```
lib: lib(ctime.o) lib(fopen.o)
    echo lib up-to-date.
```

The internal rules are already defined to complete the preceding library maintenance. The actual ".c.a" rules are as follows:

```
.c.a:
    $(CC) -c $(CFLAGS) $<
    ar rv $@ $*.o
    rm -f $*.o
```



Thus, the `$@` macro is the ".a" target (lib); the `$<` and `$*` macros are set to the out-of-date C language file; and the file name scans the suffix, respectively (*ctime.c* and *ctime*). The `$<` macro (in the preceding rule) could have been changed to `$*.c`.

It might be useful to go into some detail about exactly what **make** does when it sees the construction:

```
lib: lib (ctime.o)
@echo lib up-to-date
```

Assume the object in the library is out of date with respect to *ctime.c*. Also, there is no *ctime.o* file.

1. Do *lib*.
2. To do *lib*, do each dependent of *lib*.
3. Do *lib (ctime.o)*.
4. To do *lib (ctime.o)*, do each dependent of *lib (ctime.o)*. (There are none.)
5. Use internal rules to try to build *lib (ctime.o)*. (There is no explicit rule.) Note that *lib (ctime.o)* has a parenthesis in the name so identify the target suffix as ".a". This is the key. There is no explicit ".a" at the end of the *lib* library name. The parenthesis forces the ".a" suffix. In this sense, the ".a" is hard-wired into **make**.
6. Break the name *lib (ctime.o)* up into *lib* and *ctime.o*. Define two macros, `$@ (=lib)` and `$* (=ctime)`.
7. Look for a rule ".X.a" and a file `$*.X`. The first ".X" (in the .SUFFIXES list) which fulfills these conditions is ".c" so the rule is ".c.a" and the file is *ctime.c*. Set `$<` to be *ctime.c* and execute the rule. (In fact, **make** must then do *ctime.c*. However, the search of the current directory yields no other candidates, whence, the search ends.)
8. The library has been updated. Do the rule associated with the "lib:" dependency; namely:

```
echo lib up-to-date
```

It should be noted that to let *ctime.o* have dependencies, the following syntax is required:

```
lib(ctime.o): $(INCDIR)/stdio.h
```

Thus, explicit references to .o files are unnecessary. There is also a new macro for referencing the archive member name when this form is used. The `$%` macro is evaluated each time `$@` is evaluated. If there is no current archive member, `$%` is null. If an archive member exists, then `$%` evaluates to the expression between the parenthesis.

An example *makefile* for a larger library is given in Table 3.B. The reader will note also that there are no lingering "\*.o" files left around. The result is a library maintained directly from the source files (or more generally from the SCCS files).

#### SOURCE CODE CONTROL SYSTEM FILE NAMES: THE TILDE

The syntax of **make** does not directly permit referencing of prefixes. For most types of files on UNIX operating system machines, this is acceptable since nearly everyone uses a suffix to distinguish different types of files. The SCCS files are the exception. Here, "s." precedes the file name part of the complete pathname.



To allow *make* easy access to the prefix "s." requires either a redefinition of the rule naming syntax of *make* or a trick. The trick is to use the tilde (~) as an identifier of SCCS files. Hence, ".c~.o" refers to the rule which transforms an SCCS C language source file into an object. Specifically, the internal rule is:

```
.c~.o:
    $(GET) $(GFLAGS) -p $< > $*.c
    $(CC) $(CFLAGS) -c $*.c
    -rm -f $*.c
```

Thus, the tilde appended to any suffix transforms the file search into an SCCS file name search with the actual suffix named by the dot and all characters up to (but not including) the tilde.

The following SCCS suffixes are internally defined:

```
.c~
.y~
.s~
.sh~
.h~
```

The following rules involving SCCS transformations are internally defined:

```
.c~:
.sh~:
.c~.o:
.s~.o:
.y~.o:
.l~.o:
.y~.c:
.c~.a:
.s~.a:
.h~.h:
```

Obviously, the user can define other rules and suffixes which may prove useful. The tilde gives him a handle on the SCCS file name format so that this is possible.

#### THE NULL SUFFIX

In the UNIX system source code, there are many commands which consist of a single source file. It was wasteful to maintain an object of such files for *make's* pleasure. The current implementation supports single suffix rules (a null suffix). Thus, to maintain the program *cat*, a rule in the *makefile* of the following form is needed:

```
.c:
    $(CC) -n -O $< -o $@
```

In fact, this ".c:" rule is internally defined so no *makefile* is necessary at all. The user only needs to type:

```
make cat dd echo date
```

(these are notable single file programs) and all four C language source files are passed through the above shell command line associated with the ".c:" rule. The internally defined single suffix rules are:

```
.c:
.c~:
.sh:
.sh~:
```



Others may be added in the *makefile* by the user.

#### INCLUDE FILES

The **make** program has an include file capability. If the string *include* appears as the first seven letters of a line in a *makefile* and is followed by a blank or a tab, the following string is assumed to be a file name which the current invocation of **make** will read. The file descriptors are stacked for reading *include* files so no more than about 16 levels of nested *includes* are supported.

#### INVISIBLE SCCS MAKEFILES

The SCCS *makefiles* are invisible to **make**. That is, if **make** is typed and only a file named *s.makefile* exists, **make** will do a **get** on the file, then read and remove the file. Using the **-f**, **make** will get, read, and remove arguments and *include* files.

#### DYNAMIC DEPENDENCY PARAMETERS

A new dependency parameter has been defined. The parameter has meaning only on the dependency line in a *makefile*. The **\$\$@** refers to the current "thing" to the left of the colon (which is **\$@**). Also the form **\$\$(@F)** exists which allows access to the file part of **\$@**. Thus, in the following:

```
cat:    $$@.c
```

the dependency is translated at execution time to the string "cat.c". This is useful for building a large number of executable files, each of which has only one source file. For instance, the UNIX software command directory could have a *makefile* like:

```
CMDS = cat dd echo date cc cmp comm ar ld chown
```

```
$(CMDS):    $$@.c
            $(CC) -O $? -o $@
```

Obviously, this is a subset of all the single file programs. For multiple file programs, a directory is usually allocated and a separate *makefile* is made. For any particular file which has a peculiar compilation procedure, a specific entry must be made in the *makefile*.

The second useful form of the dependency parameter is **\$\$(@F)**. It represents the file name part of **\$\$@**. Again, it is evaluated at execution time. Its usefulness becomes evident when trying to maintain the */usr/include* directory from a *makefile* in the */usr/src/head* directory. Thus, the */usr/src/head/makefile* would look like:

```
INCDIR = /usr/include
```

```
INCLUDES = \
            $(INCDIR)/stdio.h \
            $(INCDIR)/pwd.h \
            $(INCDIR)/dir.h \
            $(INCDIR)/a.out.h
```

```
$(INCLUDES):    $$(@F)
                cp $? $@
                chmod 0444 $@
```



This would completely maintain the `/usr/include` directory whenever one of the above files in `/usr/src/head` was updated.

#### EXTENSIONS OF \$\*, \$@, AND \$<

The internally generated macros `$*`, `$@`, and `$<` are useful generic terms for current targets and out-of-date relatives. To this list has been added the following related macros: `$(*D)`, `$(*F)`, `$(<D)`, and `$(<F)`. The "D" refers to the directory part of the single letter macro. The "F" refers to the file name part of the single letter macro. These additions are useful when building hierarchical makefiles. They allow access to directory names for purposes of using the `cd` command of the shell. Thus, a shell command can be:

```
cd $(<D); $(MAKE) $(<F)
```

An interesting example of the use of these features can be found in the set of *makefiles* in Table 3.C. Each *makefile* is named "70.mk". The following command forces a complete rebuild of the operating system:

```
FRC=FRC make -f 70.mk
```

where the current directory is *uch*. The FRC is a convention for *FoRCing* `make` to completely rebuild a target starting from scratch.

#### OUTPUT TRANSLATIONS

Macros in shell commands can now be translated when evaluated. The form is as follows:

```
$(macro:string1=string2)
```

The meaning of `$(macro)` is evaluated. For each appearance of *string1* in the evaluated macro, *string2* is substituted. The meaning of finding *string1* in `$(macro)` is that the evaluated `$(macro)` is considered as a bunch of strings each delimited by white space (blanks or tabs). Thus, the occurrence of *string1* in `$(macro)` means that a regular expression of the following form has been found:

```
.*<string1>[TAB|BLANK]
```

This particular form was chosen because `make` usually concerns itself with suffixes. A more general regular expression match could be implemented if the need arises. The usefulness of this type of translation occurs when maintaining archive libraries. Now, all that is necessary is to accumulate the out-of-date members and write a shell script which can handle all the C language programs (i.e., those files ending in ".c"). Thus, the following fragment will optimize the executions of `make` for maintaining an archive library:

```
$(LIB): $(LIB)(a.o) $(LIB)(b.o) $(LIB)(c.o)
$(CC) -c $(CFLAGS) $(?:.o=.c)
ar rv $(LIB) $?
rm $?
```

A dependency of the preceding form would be necessary for each of the different types of source files (suffixes) which define the archive library. These translations are added in an effort to make more general use of the wealth of information which `make` generates.



TABLE 3.A

## EXAMPLE OF INTERNAL DEFINITIONS

```
# LIST OF SUFFIXES

.SUFFIXES: .o .c .c~ .y .y~ .l .l~ .s .s~ .sh .sh~ .h .h~

# PRESET VARIABLES

MAKE=make
YACC=yacc
YFLAGS=
LEX=lex
LFLAGS=
LD=ld
LDFLAGS=
CC=cc
CFLAGS=-O
AS=as
ASFLAGS=
GET=get
GFLAGS=

# SINGLE SUFFIX RULES

.c:
    $(CC) -n -O $< -o $@

.c~:
    $(GET) $(GFLAGS) -p $< > $*.c
    $(CC) -n -O $*.c -o $*
    -rm -f $*.c

.sh:
    cp $< $@

.sh~:
    $(GET) $(GFLAGS) -p $< > $*.sh
    cp $*.sh $*
    -rm -f $*.sh
```



TABLE 3.A (Contd)

## EXAMPLE OF INTERNAL DEFINITIONS

| #      | DOUBLE SUFFIX RULES                                                                                                                         |
|--------|---------------------------------------------------------------------------------------------------------------------------------------------|
| .c.o:  | \$(CC) \$(CFLAGS) -c \$<                                                                                                                    |
| .c~.o: | \$(GET) \$(GFLAGS) -p \$< > \$*.c<br>\$(CC) \$(CFLAGS) -c \$*.c<br>-rm -f \$*.c                                                             |
| .c~.c: | \$(GET) \$(GFLAGS) -p \$< > \$*.c                                                                                                           |
| .s.o:  | \$(AS) \$(ASFLAGS) -o \$@ \$<                                                                                                               |
| .s~.o: | \$(GET) \$(GFLAGS) -p \$< > \$*.s<br>\$(AS) \$(ASFLAGS) -o \$*.o \$*.s<br>-rm -f \$*.s                                                      |
| .y.o:  | \$(YACC) \$(YFLAGS) \$<<br>\$(CC) \$(CFLAGS) -c y.tab.c<br>rm y.tab.c<br>mv y.tab.o \$@                                                     |
| .y~.o: | \$(GET) \$(GFLAGS) -p \$< > \$*.y<br>\$(YACC) \$(YFLAGS) \$*.y<br>\$(CC) \$(CFLAGS) -c y.tab.c<br>rm -f y.tab.c \$*.y<br>mv y.tab.o \$*.o   |
| .l.o:  | \$(LEX) \$(LFLAGS) \$<<br>\$(CC) \$(CFLAGS) -c lex.yy.c<br>rm lex.yy.c<br>mv lex.yy.o \$@                                                   |
| .l~.o: | \$(GET) \$(GFLAGS) -p \$< > \$*.l<br>\$(LEX) \$(LFLAGS) \$*.l<br>\$(CC) \$(CFLAGS) -c lex.yy.c<br>rm -f lex.yy.c \$*.l<br>mv lex.yy.o \$*.o |



TABLE 3.A (Contd)

## EXAMPLE OF INTERNAL DEFINITIONS

```
.y.c:
$(YACC) $(YFLAGS) $<
mv y.tab.c $@

.y~.c:
$(GET) $(GFLAGS) -p $< > $*.y
$(YACC) $(YFLAGS) $*.y
mv y.tab.c $*.c
-rm -f $*.y

.l.c:
$(LEX) $<
mv lex.yy.c $@

.c.a:
$(CC) -c $(CFLAGS) $<
ar rv $@ $*.o
rm -f $*.o

.c~.a:
$(GET) $(GFLAGS) -p $< > $*.c
$(CC) -c $(CFLAGS) $*.c
ar rv $@ $*.o
rm -f $*.co]

.s~.a:
$(GET) $(GFLAGS) -p $< > $*.s
$(AS) $(ASFLAGS) -o $*.o $*.s
ar rv $@ $*.o
-rm -f $*.so]

.h~.h:
$(GET) $(GFLAGS) -p $< > $*.h
```



TABLE 3.8

## EXAMPLE OF LIBRARY MAKEFILE

```
#                               @(#)usr/src/cmd/make/make.tm 3.2

LIB = lsxlib

PR = vpr -b LSX

INSDIR = /r1/flop0/
INS = eval

lsx::                          $(LIB) low.o mch.o
                                ld -x low.o mch.o $(LIB)
                                mv a.out lsx
                                @size lsx

#                               Here, $(INS) as either ":" or "eval".

lsx::                          $(INS) 'cp lsx $(INSDIR)lsx . .
                                strip $(INSDIR)lsx . .
                                ls -l $(INSDIR)lsx'

print:                         $(PR) header.s low.s mch.s *.h *.c Makefile
```



TABLE 3.B (Contd)

## EXAMPLE OF LIBRARY MAKEFILE

```
$(LIB):
    $(LIB)(clock.o)
    $(LIB)(main.o)
    $(LIB)(tty.o)
    $(LIB)(trap.o)
    $(LIB)(sysent.o)
    $(LIB)(sys2.o)
    $(LIB)(sys3.o)
    $(LIB)(sys4.o)
    $(LIB)(sys1.o)
    $(LIB)(sig.o)
    $(LIB)(fio.o)
    $(LIB)(kl.o)
    $(LIB)(alloc.o)
    $(LIB)(nami.o)
    $(LIB)(iget.o)
    $(LIB)(rdwri.o)
    $(LIB)(subr.o)
    $(LIB)(bio.o)
    $(LIB)(decfd.o)
    $(LIB)(slp.o)
    $(LIB)(space.o)
    $(LIB)(puts.o)
    @echo $(LIB) now up-to-date.

.s.o:
    as -o $.o header.s $.s

.o.a:
    ar rv $@ $<
    rm -f $<

.s.a:
    as -o $.o header.s $.s
    ar rv $@ $.o
    rm -f $.o

.PRECIOUS:    $(LIB)
```



TABLE 3.C

## RECURSIVE USE OF MAKEFILES (EXAMPLE)

```
./ucb makefile

#      @(#)usr/src/cmd/make/make.tm  3.2
#      ucb/~0.mk makefile

VERSION = ~0

DEPS =      os/low.$(VERSION).o
            os/mch.$(VERSION).o
            os/conf.$(VERSION).o
            os/lib1.$(VERSION).a
            io/lib2.$(VERSION).a

#      This makefile will reload the UNIX system file
#      unix.$(VERSION) if any of the $(DEPS) is out-of-date
#      [wrt unix.$(VERSION)]. (Note: It will not go out and
#      check each member of the libraries. To do this, the FRC
#      macro must be defined.)

unix.$(VERSION):      $(DEPS) $(FRC)
                    load -s $(VERSION)

$(DEPS):              $(FRC)
                    cd $(@D); $(MAKE) -f $(VERSION).mk $(@F)

all:                  unix.$(VERSION)
                    @echo unix.$(VERSION) up-to-date.
```



TABLE 3.C (Contd)

## RECURSIVE USE OF MAKEFILES (EXAMPLE)

```
includes:
    cd head/sys; $(MAKE) -f $(VERSION).mk

FRC:    includes;
#       @(#)usr/src/cmd/make/make.tm  3.2
#       ucb/os/~0.mk makefile

VERSION = ~0

LIB = lib1.$(VERSION).a

COMPOOL =

LIBOBS =

$(LIB)(main.o)
$(LIB)(alloc.o)
$(LIB)(iget.o)
$(LIB)(prf.o)
$(LIB)(rdwri.o)
$(LIB)(slp.o)
$(LIB)(subr.o)
$(LIB)(text.o)
$(LIB)(trap.o)
$(LIB)(sig.o)
$(LIB)(sysent.o)
$(LIB)(sys1.o)
$(LIB)(sys2.o)
$(LIB)(sys3.o)
$(LIB)(sys4.o)
$(LIB)(sys5.o)
$(LIB)(syscb.o)
$(LIB)(maus.o)
$(LIB)(messag.o)
$(LIB)(nami.o)
$(LIB)(fio.o)
$(LIB)(clock.o)
$(LIB)(acct.o)
$(LIB)(errlog.o)
```



TABLE 3.C (Contd)  
RECURSIVE USE OF MAKEFILES (EXAMPLE)

```
VERSION = ~0
```

```
LIB = lib2.$(VERSION).a
```

```
COMPOOL =
```

```
LIB2OBS =
```

```
$(LIB)(mx1.o)  
$(LIB)(mx2.o)  
$(LIB)(bio.o)  
$(LIB)(tty.o)  
$(LIB)(malloc.o)  
$(LIB)(pipe.o)  
$(LIB)(dhdm.o)  
$(LIB)(dh.o)  
$(LIB)(dhfdm.o)  
$(LIB)(dj.o)  
$(LIB)(dn.o)  
$(LIB)(ds40.o)  
$(LIB)(dz.o)  
$(LIB)(alarm.o)  
$(LIB)(hf.o)  
$(LIB)(hps.o)  
$(LIB)(hpmmap.o)  
$(LIB)(hp45.o)  
$(LIB)(hs.o)  
$(LIB)(ht.o)  
$(LIB)(jy.o)  
$(LIB)(kl.o)  
$(LIB)(lfh.o)  
$(LIB)(lp.o)  
$(LIB)(mem.o)  
$(LIB)(nmpipe.o)  
$(LIB)(rf.o)  
$(LIB)(rk.o)  
$(LIB)(rp.o)  
$(LIB)(rx.o)  
$(LIB)(sys.o)  
$(LIB)(trans.o)  
$(LIB)(ttdma.o)
```



TABLE 3.C (Contd)  
RECURSIVE USE OF MAKEFILES (EXAMPLE)

```
ALL =
    conf.$(VERSION).o
    low.$(VERSION).o
    mch.$(VERSION).o
    $(LIB)

all:      $(ALL)
    @echo '$(ALL)' now up-to-date.

$(LIB)::      $(LIBOBS)

$(LIBOBS):      $(FRC);

FRC:
    rm -f $(LIB)

clobber:      cleanup
    -rm -f $(LIB)

clean cleanup:;

install:      all;

.PRECIOUS:      $(LIB)

#      @(#)/usr/src/cmd/make/make.tm 3.2
#      ucb/io/~0.mk makefile
```



TABLE 3.C (Contd)

## RECURSIVE USE OF MAKEFILES (EXAMPLE)

```
$(LIB)(tec.o)
$(LIB)(tex.o)
$(LIB)(tm.o)
$(LIB)(vp.o)
$(LIB)(vs.o)
$(LIB)(vtlp.o)
$(LIB)(vt11.o)
$(LIB)(fakevtlp.o)
$(LIB)(vt61.o)
$(LIB)(vt100.o)
$(LIB)(vtmon.o)
$(LIB)(vtdbg.o)
$(LIB)(vtutil.o)
$(LIB)(vtast.o)
$(LIB)(partab.o)
$(LIB)(rh.o)
$(LIB)(devstart.o)
$(LIB)(dmcl1.o)
$(LIB)(rop.o)
$(LIB)(ioctl.o)
$(LIB)(fakemx.o)

all:                $(LIB)
                   @echo $(LIB) is now up-to-date.

$(LIB)::            $(LIB2OBS)

$(LIB2OBS):         $(FRC)

FRC:
                   rm -f $(LIB)
```



TABLE 3.C (Contd)

## RECURSIVE USE OF MAKEFILES (EXAMPLE)

```
clobber: cleanup
        -rm -f $(LIB) *.o

clean cleanup;;

install:                                all;

.PRECIOUS:                             $(LIB)

.s.a:
        $(AS) $(ASFLAGS) -o $*.o $<
        ar rcv $@ $*.o
        rm $*.o

#      @(#)/usr/src/cmd/make/make.tm  3.2
#      ucb/head/sys/~0.mk makefile

COMPOOL = /usr/include/sys

HEADERS =

$(COMPOOL)/buf.h
$(COMPOOL)/bufx.h
$(COMPOOL)/conf.h
$(COMPOOL)/confx.h
$(COMPOOL)/crtctl.h
```



TABLE 3.C (Contd)

## RECURSIVE USE OF MAKEFILES (EXAMPLE)

```
$(COMPOOL)/dir.h
$(COMPOOL)/dm11.h
$(COMPOOL)/elog.h
$(COMPOOL)/file.h
$(COMPOOL)/filex.h
$(COMPOOL)/filsys.h
$(COMPOOL)/ino.h
$(COMPOOL)/inode.h
$(COMPOOL)/inodex.h
$(COMPOOL)/ioctl.h
$(COMPOOL)/ipcomm.h
$(COMPOOL)/ipcommx.h
$(COMPOOL)/lfs.h
$(COMPOOL)/lock.h
$(COMPOOL)/maus.h
$(COMPOOL)/mx.h
$(COMPOOL)/param.h
$(COMPOOL)/proc.h
$(COMPOOL)/procx.h
$(COMPOOL)/reg.h
$(COMPOOL)/seg.h
$(COMPOOL)/sgtty.h
$(COMPOOL)/sigdef.h
$(COMPOOL)/sprof.h
$(COMPOOL)/sprofx.h
$(COMPOOL)/stat.h
$(COMPOOL)/syserr.h
$(COMPOOL)/sysmes.h
$(COMPOOL)/sysmesx.h
$(COMPOOL)/systm.h
$(COMPOOL)/text.h
$(COMPOOL)/textx.h
$(COMPOOL)/timeb.h
$(COMPOOL)/trans.h
$(COMPOOL)/tty.h
$(COMPOOL)/ttyx.h
$(COMPOOL)/types.h
$(COMPOOL)/user.h
$(COMPOOL)/userx.h
$(COMPOOL)/version.h
$(COMPOOL)/votrax.h
$(COMPOOL)/vt11.h
$(COMPOOL)/vtmn.h
```



TABLE 3.C (Contd)

## RECURSIVE USE OF MAKEFILES (EXAMPLE)

```
all:                $(FRC) $(HEADERS)
                   @echo Headers are now up to date.

$(HEADERS):         s.$$(@F)
                   $(GET) -s -p $(GFLAGS) $? > xtemp
                   move xtemp 444 src sys $@

FRC:
                   rm -f $(HEADERS)

.PRECIOUS:          $(HEADERS)

.h~.h:
                   get -s $<

.DEFAULT:
                   cpmv $? 444 src sys $@
```



#### 4. SOURCE CODE CONTROL SYSTEM USER'S GUIDE

##### GENERAL

The Source Code Control System (SCCS) is a collection of the UNIX software commands which help individuals or projects control and account for changes to files of text. The source code and documentation of software systems are typical examples of files of text to be changed. The SCCS is a collection of programs that run under the UNIX operating system. It is convenient to conceive of SCCS as a custodian of files. The SCCS provides facilities for the following:

- Storing files of text
- Retrieving particular versions of the files
- Controlling updating privileges to files
- Identifying the version of a retrieved file
- Recording when, where, and why the change was made and who made each change to a file.

These types of facilities are important when programs and documentation undergo frequent changes because of maintenance and/or enhancement work. It is often desirable to regenerate the version of a program or document as it existed before changes were applied to it. This can be done by keeping copies (on paper or other media), but this method quickly becomes unmanageable and wasteful as the number of programs and documents increases. The SCCS provides an attractive solution because the original file is stored on disk. Whenever changes are made to the file, the SCCS stores only the changes. Each set of changes is called a "delta".

This section, together with relevant portions of the UNIX System User's Manual is a complete user's guide to SCCS. The following topics are covered:

- SCCS for Beginners: How to make an SCCS file, how to update it, and how to retrieve a version thereof.
- How Deltas Are Numbered: How versions of SCCS files are numbered and named.
- SCCS Command Conventions: Conventions and rules generally applicable to all SCCS commands.
- SCCS Commands: Explanation of all SCCS commands, with discussions of the more useful arguments.
- SCCS Files: Protection, format, and auditing of SCCS files including a discussion of the differences between using SCCS as an individual and using it as a member of a group or project. The role of a "project SCCS administrator" is introduced.

Neither the implementation of SCCS nor the installation procedure for SCCS are described in this section.

Throughout this section, each reference of the form `name(1M)`, `name(7)`, or `name(8)` refers to entries in the UNIX System Administrator's Manual. All other references to entries of the form `name(N)`, where "N" is a number (1 through 6) possibly followed by a letter, refer to entry `name` in section N of the UNIX System User's Manual.

##### SCCS FOR BEGINNERS

It is assumed that the reader knows how to log onto a UNIX system, create files, and use the text editor. A number of terminal-session fragments are presented. All of them should be tried since the best way to learn SCCS is to use it.



To supplement the material in this section, the detailed SCCS command descriptions in the UNIX System User's Manual should be consulted.

#### A. Terminology

Each SCCS file is composed of one or more sets of changes applied to the null (empty) version of the file, with each set of changes usually depending on all previous sets. Each set of changes is called a "delta" and is assigned a name, called the SCCS IDentification string (SID). The SID is composed of at most four components. The first two components are the "release" and "level" numbers which are separated by a period. Hence, the first delta (for the original file) is called "1.1", the second "1.2", the third "1.3", etc. The release number can also be changed allowing, for example, deltas "2.1", "3.1", etc. The change in the release number usually indicates a major change to the file.

Each delta of an SCCS file defines a particular version of the file. For example, delta 1.5 defines version 1.5 of the SCCS file, obtained by applying to the null (empty) version of the file the changes that constitute deltas 1.1, 1.2, etc., up to and including delta 1.5 itself, in that order.

#### B. Creating an SCCS File via "admin"

Consider, for example, a file called *lang* that contains a list of programming languages:

```
c
pl/i
fortran
cobol
algol
```

Custody of the *lang* file can be given to SCCS. The following **admin** command (used to "administer" SCCS files) creates an SCCS file and initializes delta 1.1 from the file *lang*.

```
admin -i lang s.lang
```

All SCCS files must have names that begin with "s.", hence, *s.lang*. The **-i** keyletter, together with its value *lang*, indicates that **admin** is to create a new SCCS file and "initialize" the new SCCS file with the contents of the file *lang*. This initial version is a set of changes (delta 1.1) applied to the null SCCS file.

The **admin** command replies

```
No id keywords (cm7)
```

This is a warning message (which may also be issued by other SCCS commands) that is to be ignored for the purposes of this section. Its significance is described under the **get** command in the part "SCCS COMMANDS". In the following examples, this warning message is not shown, although it may actually be issued by the various commands. The file *lang* should now be removed (because it can be easily reconstructed using the **get** command) as follows:

```
rm lang
```

#### C. Retrieving a File via "get"

The *lang* file can be reconstructed by using the following **get** command:

```
get s.lang
```



The command causes the creation (retrieval) of the latest version of file *s.lang* and prints the following messages:

```
1.1
5 lines
```

This means that *get* retrieved version 1.1 of the file, which is made up of five lines of text. The retrieved text is placed in a file whose name is formed by deleting the "s." prefix from the name of the SCCS file. Hence, the file *lang* is created.

The "get *s.lang*" command simply creates the file *lang* (read-only) and keeps no information regarding its creation. On the other hand, in order to be able to subsequently apply changes to an SCCS file with the *delta* command, the *get* command must be informed of your intention to do so. This is done as follows:

```
get -e s.lang
```

The *-e* keyletter causes *get* to create a file *lang* for both reading and writing (so it may be edited) and places certain information about the SCCS file in another new file. The new file, called the *p-file*, will be read by the *delta* command. The *get* command prints the same messages as before except that the SID of the version to be created through the use of *delta* is also issued. For example:

```
get -e s.lang
1.1
new delta 1.2
5 lines
```

The file *lang* may now be changed, for example, by:

```
ed lang
27
$a
snobol
ratfor
.
w
41
q
```

#### D. Recording Changes via "delta"

In order to record within the SCCS file the changes that have been applied to *lang*, execute the following command:

```
delta s.lang
```

Delta prompts with:

```
comments?
```

the response to which should be a description of why the changes were made. For example:

```
comments? added more languages
```

The *delta* command then reads the *p-file* and determines what changes were made to the file *lang*. The *delta* command does this by doing its own *get* to retrieve the original version and by applying the *diff(1)* command to the original version and the edited version.



When this process is complete, at which point the changes to *lang* have been stored in *s.lang*, delta outputs:

```
1.2
2 inserted
0 deleted
5 unchanged
```

The number "1.2" is the name of the delta just created, and the next three lines of output refer to the number of lines in the file *s.lang*.

#### E. Additional Information About "get"

As shown in the previous example, the command

```
get s.lang
```

retrieves the latest version (now 1.2) of the file *s.lang*. This is done by starting with the original version of the file and successively applying deltas (the changes) in order until all have been applied.

In the example chosen, the following commands are all equivalent:

```
get s.lang
get -r1 s.lang
get -r1.2 s.lang
```

The numbers following the *-r* keyletter are SIDs. Note that omitting the level number of the SID (as in "get -r1 s.lang") is equivalent to specifying the highest level number that exists within the specified release. Thus, the second command requests the retrieval of the latest version in release 1, namely 1.2. The third command specifically requests the retrieval of a particular version, in this case, also 1.2.

Whenever a truly major change is made to a file, the significance of that change is usually indicated by changing the release number (first component of the SID) of the delta being made. Since normal automatic numbering of deltas proceeds by incrementing the level number (second component of the SID), the user must indicate to SCCS the need to change the release number. This is done with the *get* command:

```
get -e -r2 s.lang
```

Because release 2 does not exist, *get* retrieves the latest version *before* release 2. The *get* command also interprets this as a request to change the release number of the delta the user desires to create to 2, thereby causing it to be named 2.1, rather than 1.3. This information is conveyed to *delta* via the *p-file*. The *get* command then outputs

```
1.2
new delta 2.1
7 lines
```

which indicates that version 1.2 has been retrieved and that 2.1 is the version *delta* will create. If the file is now edited, for example, by:

```
ed lang
41
/cobol/d
w
35
q
```



and **delta** executed:

```
delta s.lang
comments? deleted cobol from list of languages
```

the user will see by **delta**'s output that version 2.1 is indeed created:

```
2.1
0 inserted
1 deleted
6 unchanged
```

Deltas may now be created in release 2 (deltas 2.2, 2.3, etc.), or another new release may be created in a similar manner. This process may be continued as desired.

#### F. The "help" Command

If the command:

```
get abc
```

is executed, the following message will be output:

```
ERROR [abc]: not an SCCS file (col)
```

The string "col" is a code for the diagnostic message and may be used to obtain a fuller explanation of that message by use of the **help** command:

```
help col
```

This produces the following output:

```
col:
" not an SCCS file "
A file that you think is an SCCS file
does not begin with the characters "s."
```

Thus, **help** is a useful command to use whenever there is any doubt about the meaning of an SCCS message. Detailed explanations of almost all SCCS messages may be found in this manner.

#### DELTA NUMBERING

It is convenient to conceive of the deltas applied to an SCCS file as the nodes of a tree in which the root is the initial version of the file. The root delta (node) is normally named "1.1" and successor deltas (nodes) are named "1.2", "1.3", etc. The components of the names of the deltas are called the "release" and the "level" numbers, respectively. Thus, normal naming of successor deltas proceeds by incrementing the level number, which is performed automatically by SCCS whenever a delta is made. In addition, the user may wish to change the release number when making a delta to indicate that a major change is being made. When this is done, the release number also applies to all successor deltas unless specifically changed again. Thus, the evolution of a particular file may be represented as in Fig. 4.1.



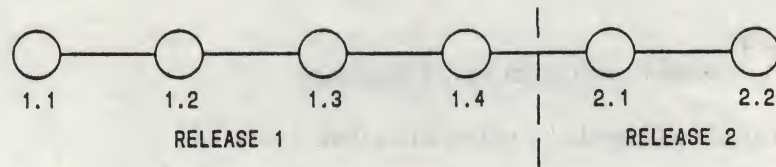


Fig. 4.1— Evolution of an SCCS File

Such a structure may be termed the “trunk” of the SCCS tree. Figure 4.1 represents the normal sequential development of an SCCS file in which changes that are part of any given delta are dependent upon all the preceding deltas.

However, there are situations in which it is necessary to cause a branching in the tree in that changes applied as part of a given delta are not dependent upon all previous deltas. As an example, consider a program which is in production use at version 1.3 and for which development work on release 2 is already in progress. Thus, release 2 may already have some deltas precisely as shown in Fig. 4.1. Assume that a production user reports a problem in version 1.3 and that the nature of the problem is such that it cannot wait to be repaired in release 2. The changes necessary to repair the trouble will be applied as a delta to version 1.3 (the version in production use). This creates a new version that will then be released to the user but will not affect the changes being applied for release 2 (i.e., deltas 1.4, 2.1, 2.2, etc.).

The new delta is a node on a branch of the tree, and its name consists of four components; the release and level numbers, as with trunk deltas, plus the “branch” and “sequence” numbers. The delta name will appear as follows:

release.level.branch.sequence

The branch number is assigned to each branch that is a descendant of a particular trunk delta with the first such branch being 1, the next one 2, etc. The sequence number is assigned, in order, to each delta on a particular branch. Thus, 1.3.1.2 identifies the second delta of the first branch that derives from delta 1.3. This is shown in Fig. 4.2.

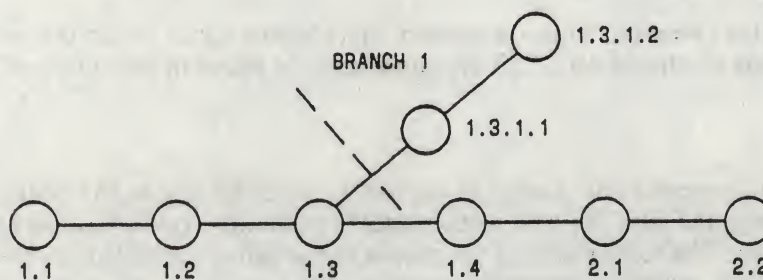


Fig. 4.2 — Tree Structure With Branch Deltas

The concept of branching may be extended to any delta in the tree. The naming of the resulting deltas proceeds in the manner just illustrated.

Two observations are of importance with regard to naming deltas. First, the names of trunk deltas contain



exactly two components, and the names of branch deltas contain exactly four components. Second, the first two components of the name of branch deltas are always those of the ancestral trunk delta, and the branch component is assigned in the order of creation of the branch independently of its location relative to the trunk delta. Thus, a branch delta may always be identified as such from its name. Although the ancestral trunk delta may be identified from the branch delta's name, it is not possible to determine the entire path leading from the trunk delta to the branch delta. For example, if delta 1.3 has one branch emanating from it, all deltas on that branch will be named 1.3.1.n. If a delta on this branch then has another branch emanating from it, all deltas on the new branch will be named 1.3.2.n (see Fig. 4.3). The only information that may be derived from the name of delta 1.3.2.2 is that it is the chronologically second delta on the chronologically second branch whose trunk ancestor is delta 1.3. In particular, it is not possible to determine from the name of delta 1.3.2.2 all the deltas between it and trunk ancestor 1.3.

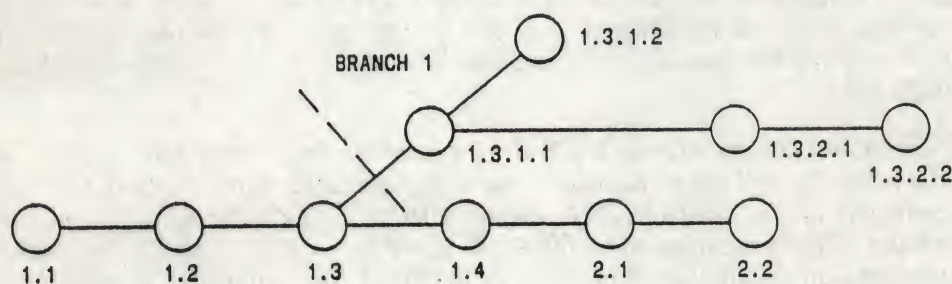


Fig. 4.3 — Extending the Branching Concept

It is obvious that the concept of branch deltas allows the generation of arbitrarily complex tree structures. Although this capability has been provided for certain specialized uses, it is strongly recommended that the SCCS tree be kept as simple as possible because comprehension of its structure becomes extremely difficult as the tree becomes more complex.

### SCCS COMMAND CONVENTIONS

This part discusses the conventions and rules that apply to SCCS commands. These rules and conventions are generally applicable to all SCCS commands with exceptions indicated. The SCCS commands accept two types of arguments:

- keyletter arguments
- file arguments.

Keyletter arguments (hereafter called simply "keyletters") begin with a minus sign (-), followed by a lower-case alphabetic character, and in some cases, followed by a value. These keyletters control the execution of the command to which they are supplied.

File arguments (which may be names of files and/or directories) specify the file(s) that the given SCCS command is to process. Naming a directory is equivalent to naming all the SCCS files within the directory. Non-SCCS files and unreadable files [because of permission modes via `chmod(1)`] in the named directories are silently ignored.

In general, file arguments may not begin with a minus sign. However, if the name "-" (a lone minus sign) is specified as an argument to a command, the command reads the standard input for lines and takes each line



as the name of an SCCS file to be processed. The standard input is read until end of file. This feature is often used in pipelines with, for example, the `find(1)` or `ls(1)` commands. Again, names of non-SCCS files and of unreadable files are silently ignored.

All keyletters specified for a given command apply to all file arguments of that command. All keyletters are processed before any file arguments with the result that the placement of keyletters is arbitrary (i.e., keyletters may be interspersed with file arguments). File arguments, however, are processed left to right. Somewhat different argument conventions apply to the `help`, `what`, `sccsdiff`, and `val` commands.

Certain actions of various SCCS commands are controlled by flags appearing in SCCS files. Some of these flags are discussed in this part. For a complete description of all such flags, see `admin(1)` section in the UNIX System User's Manual.

The distinction between the real user [see `passwd(1)`] and the effective user of a UNIX system is of concern in discussing various actions of SCCS commands. For the present, it is assumed that both the real user and the effective user are one and the same (i.e., the user who is logged into a UNIX system). This subject is discussed further in "SCCS FILES".

All SCCS commands that modify an SCCS file do so by writing a temporary copy, called the *x-file*, which ensures that the SCCS file will not be damaged should processing terminate abnormally. The name of the *x-file* is formed by replacing the "s." of the SCCS file name with "x.". When processing is complete, the old SCCS file is removed and the *x-file* is renamed to be the SCCS file. The *x-file* is created in the directory containing the SCCS file, given the same mode [see `chmod(1)`] as the SCCS file, and owned by the effective user.

To prevent simultaneous updates to an SCCS file, commands that modify SCCS files create a *lock-file*, called the *z-file*, whose name is formed by replacing the "s." of the SCCS file name with "z.". The *z-file* contains the process number of the command that creates it, and its existence is an indication to other commands that the SCCS file is being updated. Thus, other commands that modify SCCS files will not process an SCCS file if the corresponding *z-file* exists. The *z-file* is created with mode 444 (read-only) in the directory containing the SCCS file and is owned by the effective user. This file exists only for the duration of the execution of the command that creates it. In general, users can ignore *x-files* and *z-files*. The files may be useful in the event of system crashes or similar situations.

The SCCS commands produce diagnostics (on the diagnostic output) of the form:

ERROR [name-of-file-being-processed]: message text (code)

The code in parentheses may be used as an argument to the `help` command to obtain a further explanation of the diagnostic message. Detection of a fatal error during the processing of a file causes the SCCS command to terminate processing of that file and to proceed with the next file, in order, if more than one file has been named.

## SCCS COMMANDS

This part describes the major features of all the SCCS commands. Detailed descriptions of the commands and of all their arguments are given in the UNIX System User's Manual and should be consulted for further information. The discussion below covers only the more common arguments of the various SCCS commands.

The commands follow in approximate order of importance. The following is a summary of all the SCCS commands and of their major functions:

|                    |                                                                                 |
|--------------------|---------------------------------------------------------------------------------|
| <code>get</code>   | Retrieves versions of SCCS files.                                               |
| <code>delta</code> | Applies changes (deltas) to the text of SCCS files, i.e., creates new versions. |
| <code>admin</code> | Creates SCCS files and applies changes to parameters of SCCS files.             |



|                       |                                                                                                                                                                                                  |
|-----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>prs</code>      | Prints portions of an SCCS file in user specified format.                                                                                                                                        |
| <code>help</code>     | Gives explanations of diagnostic messages.                                                                                                                                                       |
| <code>rmDEL</code>    | Removes a delta from an SCCS file; allows the removal of deltas that were created by mistake.                                                                                                    |
| <code>cdc</code>      | Changes the commentary associated with a delta.                                                                                                                                                  |
| <code>what</code>     | Searches any UNIX system file(s) for all occurrences of a special pattern and prints out what follows it; is useful in finding identifying information inserted by the <code>get</code> command. |
| <code>sccsdiff</code> | Shows the differences between any two versions of an SCCS file.                                                                                                                                  |
| <code>comb</code>     | Combines two or more consecutive deltas of an SCCS file into a single delta; often reduces the size of the SCCS file.                                                                            |
| <code>val</code>      | Validates an SCCS file.                                                                                                                                                                          |

#### A. The "get" Command

The `get` command creates a text file that contains a particular version of an SCCS file. The particular version is retrieved by beginning with the initial version and then applying deltas, in order, until the desired version is obtained. The created file is called the *g-file*. The *g-file* name is formed by removing the "s." from the SCCS file name. The *g-file* is created in the current directory and is owned by the real user. The mode assigned to the *g-file* depends on how the `get` command is invoked.

The most common invocation of `get` is:

```
get s.abc
```

which normally retrieves the latest version on the trunk of the SCCS file tree and produces (for example) on the standard output:

```
1.3
67 lines
No id keywords (cm7)
```

which indicates that:

1. Version 1.3 of file "s.abc" was retrieved (1.3 is the latest trunk delta).
2. This version has 67 lines of text.
3. No ID keywords were substituted in the file.

The generated *g-file* (file "abc") is given mode 444 (read-only) since this particular way of invoking `get` is intended to produce *g-files* only for inspection, compilation, etc., and not for editing (i.e., not for making deltas).

In the case of several file arguments (or directory-name arguments), similar information is given for each file processed, but the SCCS file name precedes it. For example:

```
get s.abc s.def
```



produces:

```
s.abc:
1.3
67 lines
No id keywords (cm7)

s.def:
1.7
85 lines
No id keywords (cm7)
```

### ID Keywords

In generating a *g-file* to be used for compilation, it is useful and informative to record the date and time of creation, the version retrieved, the module's name, etc. within the *g-file*, so this information will appear in a load module when one is eventually created. The SCCS provides a convenient mechanism for doing this automatically. Identification (ID) keywords appearing anywhere in the generated file are replaced by appropriate values according to the definitions of these ID keywords. The format of an ID keyword is an uppercase letter enclosed by percent signs (%). For example:

%I%

is defined as the ID keyword that is replaced by the SID of the retrieved version of a file. Similarly, %H% is defined as the ID keyword for the current date (in the form "mm/dd/yy"), and %M% is defined as the name of the *g-file*. Thus, executing `get` on an SCCS file that contains the PL/I declaration:

```
DCL ID CHAR(100) VAR INIT('%M% %I% %H%');
```

gives (for example) the following:

```
DCL ID CHAR(100) VAR INIT('MODNAME 2.3 07/07/77');
```

When no ID keywords are substituted by `get`, the following message is issued:

```
No id keywords (cm7)
```

This message is normally treated as a warning by `get`, although the presence of the *i* flag in the SCCS file causes it to be treated as an error. For a complete list of the approximately 20 ID keywords provided, see `get(1)` in the UNIX System User's Manual.

### Retrieval of Different Versions

Various keyletters are provided to allow the retrieval of other than the default version of an SCCS file. Normally, the default version is the most recent delta of the highest-numbered release on the trunk of the SCCS file tree. However, if the SCCS file being processed has a *d* (default SID) flag, the SID specified as the value of this flag is used as a default. The default SID is interpreted in exactly the same way as the value supplied with the `-r` keyletter of `get`.

The `-r` keyletter is used to specify a SID to be retrieved, in which case the *d* (default SID) flag (if any) is ignored. For example:

```
get -r1.3 s.abc
```

retrieves version 1.3 of file *s.abc* and produces (for example) on the standard output:

```
1.3
64 lines
```



A branch delta may be retrieved similarly:

```
get -r1.5.2.3 s.abc
```

which produces (for example) on the standard output:

```
1.5.2.3  
234 lines
```

When a 2- or 4-component SID is specified as a value for the `-r` keyletter (as above) and the particular version does not exist in the SCCS file, an error message results. Omission of the level number, as in:

```
get -r3 s.abc
```

causes retrieval of the trunk delta with the highest level number within the given release if the given release exists. Thus, the above command might output:

```
3.7  
213 lines
```

If the given release does not exist, `get` retrieves the trunk delta with the highest level number within the highest-numbered existing release that is lower than the given release. For example, assuming release 9 does not exist in file `s.abc` and that release 7 is actually the highest-numbered release below 9, execution of:

```
get -r9 s.abc
```

might produce:

```
7.6  
420 lines
```

which indicates that trunk delta 7.6 is the latest version of file `s.abc` below release 9. Similarly, omission of the sequence number, as in:

```
get -r4.3.2 s.abc
```

results in the retrieval of the branch delta with the highest sequence number on the given branch if it exists. (If the given branch does not exist, an error message results.) This might result in the following output:

```
4.3.2.8  
89 lines
```

The `-t` keyletter is used to retrieve the latest (top) version in a particular release (i.e., when no `-r` keyletter is supplied or when its value is simply a release number). The latest version is defined as that delta which was produced most recently, independent of its location on the SCCS file tree. Thus, if the most recent delta in release 3 is 3.5,

```
get -r3 -t s.abc
```

might produce:

```
3.5  
59 lines
```

However, if branch delta 3.2.1.5 were the latest delta (created after delta 3.5), the same command might produce:

```
3.2.1.5  
46 lines
```



### Retrieval With Intent to Make a Delta

Specification of the `-e` keyletter to the `get` command is an indication of the intent to make a delta, and as such, its use is restricted. The presence of this keyletter causes `get` to check:

1. The user list (a list of login names and/or group IDs of users allowed to make deltas) to determine if the login name or group ID of the user executing `get` is on that list. Note that a null (empty) user list behaves as if it contained all possible login names.
2. The release (R) of the version being retrieved satisfies the relation:

floor is  $<$  or  $=$  to R which is  
 $<$  or  $=$  to ceiling

to determine if the release being accessed is a protected release. The "floor" and "ceiling" are specified as flags in the SCCS file.

3. The release (R) is not locked against editing. The "lock" is specified as a flag in the SCCS file.
4. Whether or not multiple concurrent edits are allowed for the SCCS file as specified by the `j` flag in the SCCS file.

A failure of any of the first three conditions causes the processing of the corresponding SCCS file to terminate.

If the above checks succeed, the `-e` keyletter causes the creation of a *g-file* in the current directory with mode 644 (readable by everyone, writable only by the owner) owned by the real user. If a writable *g-file* already exists, `get` terminates with an error. This is to prevent inadvertent destruction of a *g-file* that already exists and is being edited for the purpose of making a delta.

Any ID keywords appearing in the *g-file* are not substituted by `get` when the `-e` keyletter is specified because the generated *g-file* is to be subsequently used to create another delta, and replacement of ID keywords would cause them to be permanently changed within the SCCS file. In view of this, `get` does not need to check for the presence of ID keywords within the *g-file*, so the message

No id keywords (cm7)

is never output when `get` is invoked with the `-e` keyletter.

In addition, the `-e` keyletter causes the creation (or updating) of a *p-file* which is used to pass information to the `delta` command.

The following is an example of the use of the `-e` keyletter:

`get -e s.abc`

which produces (for example) on the standard output:

1.3  
new delta 1.4  
67 lines

If the `-r` and/or `-t` keyletters are used together with the `-e` keyletter, the version retrieved for editing is as specified by the `-r` and/or `-t` keyletters.



The keyletters `-i` and `-x` may be used to specify a list [see `get(1)` in the *UNIX System User's Manual* for the syntax of such a list] of deltas to be included and excluded, respectively, by `get`. Including a delta means forcing the changes that constitute the particular delta to be included in the retrieved version. This is useful if one wants to apply the same changes to more than one version of the SCCS file. Excluding a delta means forcing it to be not applied. This may be used to undo in the version of the SCCS file to be created the effects of a previous delta. Whenever deltas are included or excluded, `get` checks for possible interference between such deltas and those deltas that are normally used in retrieving the particular version of the SCCS file. Two deltas can interfere, for example, when each one changes the same line of the retrieved *g-file*. Any interference is indicated by a warning that shows the range of lines within the retrieved *g-file* in which the problem may exist. The user is expected to examine the *g-file* to determine whether a problem actually exists and to take whatever corrective measures (if any) are deemed necessary (e.g., edit the file).

**Warning:** *The `-i` and `-x` keyletters should be used with extreme care.*

The `-k` keyletter is provided to facilitate regeneration of a *g-file* that may have been accidentally removed or ruined subsequent to the execution of `get` with the `-e` keyletter or to simply generate a *g-file* in which the replacement of ID keywords has been suppressed. Thus, a *g-file* generated by the `-k` keyletter is identical to one produced by `get` executed with the `-e` keyletter. However, no processing related to the *p-file* takes place.

#### Concurrent Edits of Different SID

The ability to retrieve different versions of an SCCS file allows a number of deltas to be "in progress" at any given time. This means that a number of `get` commands with the `-e` keyletter may be executed on the same file provided that no two executions retrieve the same version (unless multiple concurrent edits are allowed).

The *p-file* (created by the `get` command invoked with the `-e` keyletter) is named by replacing the "s." in the SCCS file name with "p.". It is created in the directory containing the SCCS file, given mode 644 (readable by everyone, writable only by the owner), and owned by the effective user. The *p-file* contains the following information for each delta that is still "in progress":

- The SID of the retrieved version.
- The SID that will be given to the new delta when it is created.
- The login name of the real user executing `get`.

The first execution of `get -e` causes the creation of the *p-file* for the corresponding SCCS file. Subsequent executions only update the *p-file* with a line containing the above information. Before updating, however, `get` checks that no entry already in the *p-file* specifies as already retrieved the SID of the version to be retrieved unless multiple concurrent edits are allowed.

If both checks succeed, the user is informed that other deltas are in progress and processing continues. If either check fails, an error message results. It is important to note that the various executions of `get` should be carried out from different directories. Otherwise, only the first execution will succeed since subsequent executions would attempt to overwrite a writable *g-file*, which is an SCCS error condition. In practice, such multiple executions are performed by different users so that this problem does not arise since each user normally has a different working directory. See "Protection" under the part "SCCS FILES" for a discussion of how different users are permitted to use SCCS commands on the same files.

Table 4.A shows, for the most useful cases, the version of an SCCS file retrieved by `get`, as well as the SID of the version to be eventually created by `delta`, as a function of the SID specified to `get`.

#### Concurrent Edits of Same SID

Under normal conditions, `gets` for editing (`-e` keyletter is specified) based on the same SID are not permitted to occur concurrently. That is, `delta` must be executed before a subsequent `get` for editing is executed at



the same SID as the previous **get**. However, multiple concurrent edits (defined to be two or more successive executions of **get** for editing based on the same retrieved SID) are allowed if the **j** flag is set in the SCCS file. Thus:

```
get -e s.abc
1.1
new delta 1.2
5 lines
```

may be immediately followed by:

```
get -e s.abc
1.1
new delta 1.1.1.1
5 lines
```

without an intervening execution of **delta**. In this case, a **delta** command corresponding to the first **get** produces delta 1.2 (assuming 1.1 is the latest (most recent) trunk delta), and the **delta** command corresponding to the second **get** produces delta 1.1.1.1.

#### Keyletters That Affect Output

Specification of the **-p** keyletter causes **get** to write the retrieved text to the standard output rather than to a *g-file*. In addition, all output normally directed to the standard output (such as the SID of the version retrieved and the number of lines retrieved) is directed instead to the diagnostic output. This may be used, for example, to create *g-files* with arbitrary names:

```
get -p s.abc > arbitrary-file-name
```

The **-p** keyletter is particularly useful when used with the **!"** or **!"** arguments of the **send(1C)** command. For example:

```
send MOD=s.abc REL=3 compile
```

given that file *compile* contains:

```
//plicomp job job-card-information
//stepl exec plicke
//pli.sysin dd *
~s
~!get -p -rREL MOD
/*
//
```

will **send** the highest level of release 3 of file *s.abc*. Note that the line **~s**, which causes **send** to make ID keyword substitutions before detecting and interpreting control lines, is necessary if **send** is to substitute **"s.abc"** for **MOD** and **"3"** for **REL** in the line **~!get -p -rREL MOD**.

The **-s** keyletter suppresses all output that is normally directed to the standard output. Thus, the SID of the retrieved version, the number of lines retrieved, etc., are not output. This does not, however, affect messages to the diagnostic output. This keyletter is used to prevent nondiagnostic messages from appearing on the user's terminal and is often used in conjunction with the **-p** keyletter to "pipe" the output of **get**, as in:

```
get -p -s s.abc | nroff
```



The `-g` keyletter is supplied to suppress the actual retrieval of the text of a version of the SCCS file. This may be useful in a number of ways. For example, to verify the existence of a particular SID in an SCCS file, one may execute:

```
get -g -r4.3 s.abc
```

This outputs the given SID if it exists in the SCCS file or it generates an error message if it does not. Another use of the `-g` keyletter is in regenerating a *p-file* that may have been accidentally destroyed:

```
get -e -g s.abc
```

The `-l` keyletter causes the creation of an *l-file*, which is named by replacing the "s." of the SCCS file name with "l.". This file is created in the current directory with mode 444 (read-only) and is owned by the real user. It contains a table (whose format is described in `get(1)` in the UNIX System User's Manual) showing which deltas were used in constructing a particular version of the SCCS file. For example:

```
get -r2.3 -l s.abc
```

generates an *l-file* showing the deltas applied to retrieve version 2.3 of the SCCS file. Specifying a value of "p" with the `-l` keyletter, as in:

```
get -lp -r2.3 s.abc
```

causes the generated output to be written to the standard output rather than to the *l-file*. The `-g` keyletter may be used with the `-l` keyletter to suppress the actual retrieval of the text.

The `-m` keyletter is of use in identifying, line by line, the changes applied to an SCCS file. Specification of this keyletter causes each line of the generated *g-file* to be preceded by the SID of the delta that caused that line to be inserted. The SID is separated from the text of the line by a tab character.

The `-n` keyletter causes each line of the generated *g-file* to be preceded by the value of the `%M%` ID keyword and a tab character. The `-n` keyletter is most often used in a pipeline with `grep(1)`. For example, to find all lines that match a given pattern in the latest version of each SCCS file in a directory, the following may be executed:

```
get -p -n -s directory | grep pattern
```

If both the `-m` and `-n` keyletters are specified, each line of the generated *g-file* is preceded by the value of the `%M%` ID keyword and a tab (this is the effect of the `-n` keyletter) and followed by the line in the format produced by the `-m` keyletter. Because use of the `-m` keyletter and/or the `-n` keyletter causes the contents of the *g-file* to be modified, such a *g-file* must not be used for creating a delta. Therefore, neither the `-m` keyletter nor the `-n` keyletter may be specified together with the `-e` keyletter.

See `get(1)` in the UNIX System User's Manual for a full description of additional `get` keyletters.

## B. The "delta" Command

The `delta` command is used to incorporate the changes made to a *g-file* into the corresponding SCCS file, i.e., to create a delta, and therefore, a new version of the file.

Invocation of the `delta` command requires the existence of a *p-file*. The `delta` command examines the *p-file* to verify the presence of an entry containing the user's login name. If none is found, an error message results. The `delta` command also performs the same permission checks that `get` performs when invoked with the `-e`



keyletter. If all checks are successful, **delta** determines what has been changed in the *g-file* by comparing it via `diff(1)` with its own temporary copy of the *g-file* as it was before editing. This temporary copy of the *g-file* is called the *d-file* (its name is formed by replacing the "s." of the SCCS file name with "d.") and is obtained by performing an internal `get` at the SID specified in the *p-file* entry.

The required *p-file* entry is the one containing the login name of the user executing **delta** because the user who retrieved the *g-file* must be the one who will create the delta. However, if the login name of the user appears in more than one entry (i.e., the same user executed `get` with the `-e` keyletter more than once on the same SCCS file), the `-r` keyletter must be used with **delta** to specify an SID that uniquely identifies the *p-file* entry. This entry is the one used to obtain the SID of the delta to be created.

In practice, the most common invocation of **delta** is

```
delta s.abc
```

which prompts on the standard output (but only if it is a terminal):

```
comments?
```

to which the user replies with a description of why the delta is being made, terminating the reply with a new line character. The user's response may be up to 512 characters long with new lines, not intended to terminate the response, escaped by backslash "\".

If the SCCS file has a `v` flag, **delta** first prompts with

```
MRs?
```

on the standard output. (Again, this prompt is printed only if the standard output is a terminal.) The standard input is then read for MR numbers, separated by blanks and/or tabs, terminated in the same manner as the response to the prompt "comments?". In a tightly controlled environment, it is expected that deltas are created only as a result of some trouble report, change request, trouble ticket, etc. (collectively called here Modification Requests [MRs]) and that it is desirable or necessary to record such MR number(s) within each delta.

The `-y` and/or `-m` keyletters may be used to supply the commentary (comments and MR numbers, respectively) on the command line rather than through the standard input:

```
delta -y "descriptive comment" -m "mrnum1 mrnum2" s.abc
```

In this case, the corresponding prompts are not printed, and the standard input is not read. The `-m` keyletter is allowed only if the SCCS file has a `v` flag. These keyletters are useful when **delta** is executed from within a shell procedure [see `sh(1)` in the UNIX System User's Manual.]

The commentary (comments and/or MR numbers), whether solicited by **delta** or supplied via keyletters, is recorded as part of the entry for the delta being created and applies to all SCCS files processed by the same invocation of **delta**. This implies that if **delta** is invoked with more than one file argument and the first file named has a `v` flag all files named must have this flag. Similarly, if the first file named does not have this flag, then none of the files named may have it. Any file that does not conform to these rules is not processed.

When processing is complete, **delta** outputs (on the standard output) the SID of the created delta (obtained from the *p-file* entry) and the counts of lines inserted, deleted, and left unchanged by the delta. Thus, a typical output might be:

```
1.4
14 inserted
7 deleted
345 unchanged
```



It is possible that the counts of lines reported as inserted, deleted, or unchanged by **delta** do not agree with the user's perception of the changes applied to the *g-file*. The reason for this is that there usually are a number of ways to describe a set of such changes, especially if lines are moved around in the *g-file*, and **delta** is likely to find a description that differs from the user's perception. However, the total number of lines of the new delta (the number inserted plus the number left unchanged) should agree with the number of lines in the edited *g-file*.

If in the process of making a delta **delta** finds no ID keywords in the edited *g-file*, the message

No id keywords (cm7)

is issued after the prompts for commentary but before any other output. This indicates that any ID keywords that may have existed in the SCCS file have been replaced by their values or deleted during the editing process. This could be caused by creating a delta from a *g-file* that was created by a **get** without the **-e** keyletter (recall that ID keywords are replaced by **get** in that case) or by accidentally deleting or changing the ID keywords during the editing of the *g-file*. Another possibility is that the file may never have had any ID keywords. In any case, it is left up to the user to determine what remedial action is necessary, but the delta is made, unless there is an **i** flag in the SCCS file indicating that this should be treated as a fatal error. In this last case, the delta is not created.

After processing of an SCCS file is complete, the corresponding *p-file* entry is removed from the *p-file*. All updates to the *p-file* are made to a temporary copy, the *q-file*, whose use is similar to the use of the *x-file* which is described in the part "SCCS COMMAND CONVENTIONS". If there is only one entry in the *p-file*, then the *p-file* itself is removed.

In addition, **delta** removes the edited *g-file* unless the **-n** keyletter is specified. Thus:

**delta -n s.abc**

will keep the *g-file* upon completion of processing.

The **-s** (silent) keyletter suppresses all output that is normally directed to the standard output, other than the prompts "comments?" and "MRs?". Thus, use of the **-s** keyletter together with the **-y** keyletter (and possibly, the **-m** keyletter) causes **delta** neither to read the standard input nor to write the standard output.

The differences between the *g-file* and the *d-file* (see above), which constitute the delta, may be printed on the standard output by using the **-p** keyletter. The format of this output is similar to that produced by **diff(1)**.

### C. The "admin" Command

The **admin** command is used to administer SCCS files, that is, to create new SCCS files and to change parameters of existing ones. When an SCCS file is created, its parameters are initialized by use of keyletters or are assigned default values if no keyletters are supplied. The same keyletters are used to change the parameters of existing files.

Two keyletters are supplied for use in conjunction with detecting and correcting "corrupted" SCCS files. (Discussed in "Auditing" under the part "SCCS FILES".) Newly created SCCS files are given mode 444 (read-only) and are owned by the effective user. Only a user with write permission in the directory containing the SCCS file may use the **admin** command upon that file.

#### Creation of SCCS Files

An SCCS file may be created by executing the command

**admin -ifirst s.abc**



in which the value "first" of the `-i` keyletter specifies the name of a file from which the text of the initial delta of the SCCS file `s.abc` is to be taken. Omission of the value of the `-i` keyletter indicates that `admin` is to read the standard input for the text of the initial delta. Thus, the command

```
admin -i s.abc < first
```

is equivalent to the previous example. If the text of the initial delta does not contain ID keywords, the message

```
No id keywords (cm7)
```

is issued by `admin` as a warning. However, if the same invocation of the command also sets the `i` flag (not to be confused with the `-i` keyletter), the message is treated as an error and the SCCS file is not created. Only one SCCS file may be created at a time using the `-i` keyletter.

When an SCCS file is created, the release number assigned to its first delta is normally "1", and its level number is always "1". Thus, the first delta of an SCCS file is normally "1.1". The `-r` keyletter is used to specify the release number to be assigned to the first delta. Thus:

```
admin -ifirst -r3 s.abc
```

indicates that the first delta should be named "3.1" rather than "1.1". Because this keyletter is only meaningful in creating the first delta, its use is only permitted with the `-i` keyletter.

#### Inserting Commentary for the Initial Delta

When an SCCS file is created, the user may choose to supply commentary stating the reason for creation of the file. This is done by supplying comments (`-y` keyletter) and/or MR numbers (`-m` keyletter) in exactly the same manner as for delta. The creation of an SCCS file may sometimes be the direct result of an MR. If comments (`-y` keyletter) are omitted, a comment line of the form

```
date and time created YY/MM/DD HH:MM:SS by logname
```

is automatically generated.

If it is desired to supply MR numbers (`-m` keyletter), the `v` flag must also be set (using the `-f` keyletter described below). The `v` flag simply determines whether or not MR numbers must be supplied when using any SCCS command that modifies a "delta commentary" [see `sccsfile(4)` in the UNIX System User's Manual] in the SCCS file. Thus:

```
admin -ifirst -mmrnum1 -fv s.abc
```

Note that the `-y` and `-m` keyletters are only effective if a new SCCS file is being created.

#### Initialization and Modification of SCCS File Parameters

The portion of the SCCS file reserved for descriptive text may be initialized or changed through the use of the `-t` keyletter. The descriptive text is intended as a summary of the contents and purpose of the SCCS file.

When an SCCS file is being created and the `-t` keyletter is supplied, it must be followed by the name of a file from which the descriptive text is to be taken. For example, the command

```
admin -ifirst -tdesc s.abc
```



specifies that the descriptive text is to be taken from file *desc*.

When processing an *existing* SCCS file, the `-t` keyletter specifies that the descriptive text (if any) currently in the file is to be replaced with the text in the named file. Thus:

```
admin -tdesc s.abc
```

specifies that the descriptive text of the SCCS file is to be replaced by the contents of *desc*; omission of the file name after the `-t` keyletter as in

```
admin -t s.abc
```

causes the removal of the descriptive text from the SCCS file.

The flags of an SCCS file may be initialized, changed, or deleted through the use of the `-f` and `-d` keyletters, respectively. The flags of an SCCS file are used to direct certain actions of the various commands. See `admin(1)` in the UNIX System User's Manual for a description of all the flags. For example, the `i` flag specifies that the warning message stating there are no ID keywords contained in the SCCS file should be treated as an error, and the `d` (default SID) flag specifies the default version of the SCCS file to be retrieved by the `get` command. The `-f` keyletter is used to set a flag and, possibly, to set its value. For example:

```
admin -ifirst -fi -fmmodname s.abc
```

sets the `i` flag and the `m` (module name) flag. The value "modname" specified for the `m` flag is the value that the `get` command will use to replace the `%M%` ID keyword. (In the absence of the `m` flag, the name of the *g-file* is used as the replacement for the `%M%` ID keyword.) Note that several `-f` keyletters may be supplied on a single invocation of `admin` and that `-f` keyletters may be supplied whether the command is creating a new SCCS file or processing an existing one.

The `-d` keyletter is used to delete a flag from an SCCS file and may only be specified when processing an existing file. As an example, the command

```
admin -dm s.abc
```

removes the `m` flag from the SCCS file. Several `-d` keyletters may be supplied on a single invocation of `admin` and may be intermixed with `-f` keyletters.

The SCCS files contain a list (user list) of login names and/or group IDs of users who are allowed to create deltas. This list is empty by default which implies that anyone may create deltas. To add login names and/or group IDs to the list, the `-a` keyletter is used. For example:

```
admin -axyz -awql -a1234 s.abc
```

adds the login names "xyz" and "wql" and the group ID "1234" to the list. The `-a` keyletter may be used whether `admin` is creating a new SCCS file or processing an existing one and may appear several times. The `-e` keyletter is used in an analogous manner if one wishes to remove (erase) login names or group IDs from the list.

#### D. The "prs" Command

The `prs` command is used to print on the standard output all or parts of an SCCS file in a format, called the output "data specification," supplied by the user via the `-d` keyletter. The data specification is a string consisting of SCCS file data keywords (not to be confused with `get` ID keywords) interspersed with optional user text.

Data keywords are replaced by appropriate values according to their definitions. For example:

```
:I:
```



is defined as the data keyword that is replaced by the SID of a specified delta. Similarly, :F: is defined as the data keyword for the SCCS file name currently being processed, and :C: is defined as the comment line associated with a specified delta. All parts of an SCCS file have an associated data keyword. For a complete list of the data keywords, see `prs(1)` in the UNIX System User's Manual.

There is no limit to the number of times a data keyword may appear in a data specification. Thus, for example:

```
prs -d ":I: this is the top delta for :F: :I:" s.abc
```

may produce on the standard output

```
2.1 this is the top delta for s.abc 2.1
```

Information may be obtained from a single delta by specifying the SID of that delta using the `-r` keyletter. For example:

```
prs -d ":F: : :I: comment line is: :C:" -r1.4 s.abc
```

may produce the following output:

```
s.abc: 1.4 comment line is: THIS IS A COMMENT
```

If the `-r` keyletter is not specified, the value of the SID defaults to the most recently created delta.

In addition, information from a range of deltas may be obtained by specifying the `-l` or `-e` keyletters. The `-e` keyletter substitutes data keywords for the SID designated via the `-r` keyletter and all deltas created earlier. The `-l` keyletter substitutes data keywords for the SID designated via the `-r` keyletter and all deltas created later. Thus, the command

```
prs -d:I: -r1.4 -e s.abc
```

may output

```
1.4  
1.3  
1.2.1.1  
1.2  
1.1
```

and the command

```
prs -d:I: -r1.4 -l s.abc
```

may produce

```
3.3  
3.2  
3.1  
2.2.1.1  
2.2  
2.1  
1.4
```



Substitution of data keywords for all deltas of the SCCS file may be obtained by specifying both the `-e` and `-l` keyletters.

#### E. The "help" Command

The `help` command prints explanations of SCCS commands and of messages that these commands may print. Arguments to `help`, zero or more of which may be supplied, are simply the names of SCCS commands or the code numbers that appear in parentheses after SCCS messages. If no argument is given, `help` prompts for one. The `help` command has no concept of keyletter arguments or file arguments. Explanatory information related to an argument, if it exists, is printed on the standard output. If no information is found, an error message is printed. Note that each argument is processed independently, and an error resulting from one argument will not terminate the processing of the other arguments.

Explanatory information related to a command is a synopsis of the command. For example:

```
help ge5 rmdel
```

produces

```
ge5:
" nonexistent sid "
The specified sid does not exist in the
given file.
Check for typos.

rmdel:
rmdel -rSID name ...
```

#### F. The "rmdel" Command

The `rmdel` command is provided to allow removal of a delta from an SCCS file. Its use should be reserved for those cases in which incorrect global changes were made a part of the delta to be removed.

The delta to be removed must be a "leaf" delta. That is, it must be the latest (most recently created) delta on its branch or on the trunk of the SCCS file tree. In Fig. 4.3, only deltas 1.3.1.2, 1.3.2.2, and 2.2 can be removed; once they are removed then deltas 1.3.2.1 and 2.1 can be removed, etc.

To be allowed to remove a delta, the effective user must have write permission in the directory containing the SCCS file. In addition, the real user must either be the one who created the delta being removed or be the owner of the SCCS file and its directory.

The `-r` keyletter, which is mandatory, is used to specify the complete SID of the delta to be removed (i.e., it must have two components for a trunk delta and four components for a branch delta). Thus:

```
rmdel -r2.3 s.abc
```

specifies the removal of (trunk) delta "2.3" of the SCCS file. Before removal of the delta, `rmdel` checks that the release number (R) of the given SID satisfies the relation:

$$\text{floor} \leq R \leq \text{ceiling}$$



The `rmDEL` command also checks that the SID specified is not that of a version for which a `get` for editing has been executed and whose associated `delta` has not yet been made. In addition, the login name or group ID of the user must appear in the file's "user list", or the "user list" must be empty. Also, the release specified can not be locked against editing. That is, if the `l` flag is set [see `admin(1)` in the UNIX System User's Manual], the release specified *must* not be contained in the list. If these conditions are not satisfied, processing is terminated, and the `delta` is not removed. After the specified `delta` has been removed, its type indicator in the "delta table" of the SCCS file is changed from "D" (`delta`) to "R" (`removed`).

#### G. The "`cdc`" Command

The `cdc` command is used to change a `delta`'s commentary that was supplied when that `delta` was created. Its invocation is analogous to that of the `rmDEL` command, except that the `delta` to be processed is not required to be a leaf `delta`. For example:

```
cdc -r3.4 s.abc
```

specifies that the commentary of `delta` "3.4" of the SCCS file is to be changed.

The new commentary is solicited by `cdc` in the same manner as that of `delta`. The old commentary associated with the specified `delta` is kept, but it is preceded by a comment line indicating that it has been changed (i.e., superseded), and the new commentary is entered ahead of this comment line. The "inserted" comment line records the login name of the user executing `cdc` and the time of its execution.

The `cdc` command also allows for the deletion of selected MR numbers associated with the specified `delta`. This is specified by preceding the selected MR numbers by the character "!". Thus:

```
cdc -r1.4 s.abc
MRs? mrnum3 !mrnum1
comments? deleted wrong MR number and inserted correct MR number
```

inserts "mrnum3" and deletes "mrnum1" for `delta` 1.4.

#### H. The "`what`" Command

The `what` command is used to find identifying information within any UNIX system file whose name is given as an argument to `what`. Directory names and a name of "-" (a lone minus sign) are not treated specially, as they are by other SCCS commands, and no keyletters are accepted by the command.

The `what` command searches the given file(s) for all occurrences of the string "@(#)", which is the replacement for the %Z% ID keyword [see `get(1)`], and prints (on the standard output) what follows that string until the first double quote ( " ), greater than (>), backslash (\), new line, or (nonprinting) NUL character. For example, if the SCCS file `s.prog.c` (a C language program) contains the following line:

```
char id[] " %Z% %M%:%I%";
```

and then the command

```
get -r3.4 s.prog.c
```

is executed, the resulting *g-file* is compiled to produce "`prog.o`" and "`a.out`". Then the command

```
what prog.c prog.o a.out
```



produces

```
prog.c:
  prog.c:3.4
prog.o:
  prog.c:3.4
a.out:
  prog.c:3.4
```

The string searched for by **what** need not be inserted via an ID keyword of **get**; it may be inserted in any convenient manner.

#### I. The "sccsdiff" Command

The **sccsdiff** command determines (and prints on the standard output) the differences between two specified versions of one or more SCCS files. The versions to be compared are specified by using the **-r** keyletter, whose format is the same as for the **get** command. The two versions must be specified as the first two arguments to this command in the order they were created, i.e., the older version is specified first. Any following keyletters are interpreted as arguments to the **pr(1)** command (which actually prints the differences) and must appear before any file names. The SCCS files to be processed are named last. Directory names and a name of "-" (a lone minus sign) are not acceptable to **sccsdiff**.

The differences are printed in the form generated by **diff(1)**. The following is an example of the invocation of **sccsdiff**:

```
sccsdiff -r3.4 -r5.6 s.abc
```

#### J. The "comb" Command

The **comb** command generates a "shell procedure" [see **sh(1)** in the UNIX System User's Manual] which attempts to reconstruct the named SCCS files so that the reconstructed files are smaller than the originals. The generated shell procedure is written on the standard output. Named SCCS files are reconstructed by discarding unwanted deltas and combining other specified deltas. The SCCS files that contain deltas no longer useful should be discarded. It is not recommended that **comb** be used as a matter of routine; its use should be restricted to a very small number of times in the life of an SCCS file.

In the absence of any keyletters, **comb** preserves only leaf deltas and the minimum number of ancestor deltas necessary to preserve the "shape" of the SCCS file tree. The effect of this is to eliminate middle deltas on the trunk and on all branches of the tree. Thus, in Fig. 4.3, deltas 1.2, 1.3.2.1, 1.4, and 2.1 would be eliminated. Some of the keyletters are summarized as follows:

The **-p** keyletter specifies the oldest delta that is to be preserved in the reconstruction. All older deltas are discarded.

The **-c** keyletter specifies a list [see **get(1)** in the UNIX System User's Manual for the syntax of such a list] of deltas to be preserved. All other deltas are discarded.

The **-s** keyletter causes the generation of a shell procedure, which when run, produces only a report summarizing the percentage space (if any) to be saved by reconstructing each named SCCS file. It is recommended that **comb** be run with this keyletter (in addition to any others desired) before any actual reconstructions.

It should be noted that the shell procedure generated by **comb** is not guaranteed to save space. In fact, it is possible for the reconstructed file to be larger than the original. Note, too, that the shape of the SCCS file tree may be altered by the reconstruction process.



### K. The "val" Command

The **val** command is used to determine if a file is an SCCS file meeting the characteristics specified by an optional list of keyletter arguments. Any characteristics not met are considered errors.

The **val** command checks for the existence of a particular delta when the SID for that delta is explicitly specified via the **-r** keyletter. The string following the **-y** or **-m** keyletter is used to check the value set by the **t** or **m** flag, respectively [see **admin(1)** in the UNIX System User's Manual for a description of the flags].

The **val** command treats the special argument **"-"** differently from other SCCS commands. This argument allows **val** to read the argument list from the standard input as opposed to obtaining it from the command line. The standard input is read until end of file. This capability allows for one invocation of **val** with different values for the keyletter and file arguments. For example:

```
val -  
-yc -mabc s.abc  
-mxyz -ypl1 s.xyz
```

first checks if file *s.abc* has a value **"c"** for its **"type"** flag and value **"abc"** for the **"module name"** flag. Once processing of the first file is completed, **val** then processes the remaining files, in this case, *s.xyz*, to determine if they meet the characteristics specified by the keyletter arguments associated with them.

The **val** command returns an 8-bit code; each bit set indicates the occurrence of a specific error [see **val(1)** for a description of possible errors and the codes]. In addition, an appropriate diagnostic is printed unless suppressed by the **-s** keyletter. A return code of **"0"** indicates all named files met the characteristics specified.

### SCCS FILES

This part discusses several topics that must be considered before extensive use is made of SCCS. These topics deal with the protection mechanisms relied upon by SCCS, the format of SCCS files, and the recommended procedures for auditing SCCS files.

#### A. Protection

The SCCS relies on the capabilities of the UNIX software for most of the protection mechanisms required to prevent unauthorized changes to SCCS files (i.e., changes made by non-SCCS commands). The only protection features provided directly by SCCS are the **"release lock"** flag, the **"release floor"** and **"ceiling"** flags, and the **"user list"**.

New SCCS files created by the **admin** command are given mode 444 (read-only). It is recommended that this mode *not* be changed as it prevents any direct modification of the files by non-SCCS commands. It is further recommended that the directories containing SCCS files be given mode 755 which allows only the owner of the directory to modify its contents.

The SCCS files should be kept in directories that contain only SCCS files and any temporary files created by SCCS commands. This simplifies protection and auditing of SCCS files. The contents of directories should correspond to convenient logical groupings, e.g., subsystems of a large project.

The SCCS files must have only one link (name) because the commands that modify SCCS files do so by creating a copy of the file (the *x-file*, see **"SCCS COMMAND CONVENTIONS"**) and, upon completion of processing, remove the old file and rename the *x-file*. If the old file has more than one link, this would break such additional links. Rather than process such files, SCCS commands produce an error message. All SCCS files *must* have names that begin with **"s."**



When only one user uses SCCS, the real and effective user IDs are the same; and the user ID owns the directories containing SCCS files. Therefore, SCCS may be used directly without any preliminary preparation.

However, in those situations in which several users with unique user IDs are assigned responsibility for one SCCS file (e.g., in large software development projects), one user (equivalently, one user ID) must be chosen as the "owner" of the SCCS files and be the one who will "administer" them (e.g., by using the **admin** command). This user is termed the "SCCS administrator" for that project. Because other users of SCCS do not have the same privileges and permissions as the SCCS administrator, they are not able to execute directly those commands that require write permission in the directory containing the SCCS files. Therefore, a project-dependent program is required to provide an interface to the **get**, **delta**, and if desired, **rmDEL** and **cdc** commands.

The interface program must be owned by the SCCS administrator and must have the "set user ID on execution" bit "on" [see **chmod(1)** in the UNIX System User's Manual], so that the effective user ID is the user ID of the administrator. This program invokes the desired SCCS command and causes it to inherit the privileges of the interface program for the duration of that command's execution. Thus, the owner of an SCCS file can modify it at will. Other users whose login names or group IDs are in the "user list" for that file (but are not the owner) are given the necessary permissions only for the duration of the execution of the interface program. These other users are thus able to modify the SCCS files only through the use of **delta** and, possibly, **rmDEL** and **cdc**. The project-dependent interface program, as its name implies, must be custom-built for each project.

#### B. Formatting

The SCCS files are composed of lines of ASCII text arranged in six parts as follows:

|                  |                                                                                                                     |
|------------------|---------------------------------------------------------------------------------------------------------------------|
| Checksum         | A line containing the "logical" sum of all the characters of the file ( <i>not</i> including this checksum itself). |
| Delta Table      | Information about each delta, such as type, SID, date and time of creation, and commentary.                         |
| User Names       | List of login names and/or group IDs of users who are allowed to modify the file by adding or removing deltas.      |
| Flags            | Indicators that control certain actions of various SCCS commands.                                                   |
| Descriptive Text | Arbitrary text provided by the user; usually a summary of the contents and purpose of the file.                     |
| Body             | Actual text that is being administered by SCCS, intermixed with internal SCCS control lines.                        |

Detailed information about the contents of the various sections of the file may be found in **scsfile(5)**. The checksum is the only portion of the file which is of interest below.

It is important to note that because SCCS files are ASCII files they may be processed by various UNIX software commands, such as **ed(1)**, **grep(1)**, and **cat(1)**. This is very convenient in those instances in which an SCCS file must be modified manually (e.g., when the time and date of a delta was recorded incorrectly because the system clock was set incorrectly) or when it is desired to simply look at the file.

**Caution:** *Extreme care should be exercised when modifying SCCS files with non-SCCS commands.*

#### C. Auditing

On rare occasions, perhaps due to an operating system or hardware malfunction, an SCCS file or portions of it (i.e., one or more "blocks") can be destroyed. The SCCS commands (like most UNIX software commands)



issue an error message when a file does not exist. In addition, SCCS commands use the checksum stored in the SCCS file to determine whether a file has been corrupted since it was last accessed [possibly by having lost one or more blocks or by having been modified with `ed(1)`]. No SCCS command will process a corrupted SCCS file except the `admin` command with the `-h` or `-z` keyletters, as described below.

It is recommended that SCCS files be audited for possible corruptions on a regular basis. The simplest and fastest way to perform an audit is to execute the `admin` command with the `-h` keyletter on all SCCS files:

```
admin -h s.file1 s.file2 ...  
      or  
admin -h directory1 directory2 ...
```

If the new checksum of any file is not equal to the checksum in the first line of that file, the message

```
corrupted file (co6)
```

is produced for that file. This process continues until all the files have been examined. When examining directories (as in the second example above), the process just described will not detect missing files. A simple way to detect whether any files are missing from a directory is to periodically execute the `ls(1)` command on that directory and compare the outputs of the most current and the previous executions. Any file whose name appears in the previous output but not in the current one has been removed by some means.

Whenever a file has been corrupted, the manner the file is restored depends upon the extent of the corruption. If damage is extensive, the best solution is to contact the local UNIX system operations group and request the file be restored from a backup copy. In the case of minor damage, repair through use of the editor `ed(1)` may be possible. In the latter case after such repair, the following command must be executed:

```
admin -z s.file
```

The purpose of this is to recompute the checksum to bring it into agreement with the actual contents of the file. After this command is executed on a file, any corruption that existed in the file will no longer be detectable.

## AN SCCS INTERFACE PROGRAM

### A. General

In order to permit UNIX system users with different user identification numbers (user IDs) to use SCCS commands upon the same files, an SCCS interface program is provided to temporarily grant the necessary file access permissions to these users. This part discusses the creation and use of such an interface program. The SCCS interface program may also be used as a preprocessor to SCCS commands since it can perform operations upon its arguments.

### B. Function

When only one user uses SCCS, the real and effective user IDs are the same; and that user's ID owns the directories containing SCCS files. However, there are situations (e.g., in large software development projects) in which it is practical to allow more than one user to make changes to the same set of SCCS files. In these cases, one user must be chosen as the "owner" of the SCCS files and be the one who will "administer" them (e.g., by using the `admin` command). This user is termed the "SCCS administrator" for that project. Since other users of SCCS do not have the same privileges and permissions as the SCCS administrator, the other users are not able to execute directly those commands that require write permission in the directory containing the SCCS files. Therefore, a project-dependent program is required to provide an interface to the `get`, `delta`, and if desired, `rmDEL`, `cdc`, and `unget` commands. Other SCCS commands either do not require write permission in the directory containing SCCS files or are (generally) reserved for use only by the administrator.



The interface program must be owned by the SCCS administrator, must be executable by nonowners, and must have the "set user ID on execution" bit "on" [see `chmod(1)` in the UNIX System User's Manual] so that, when executed, the effective user ID is the user ID of the administrator. This program's function is to invoke the desired SCCS command and to cause it to inherit the privileges of the SCCS administrator for the duration of that command's execution. In this manner, the owner of an SCCS file (the administrator) can modify it at will. Other users whose login names are in the user list for that file (but who are not its owners) are given the necessary permissions only for the duration of the execution of the interface program. They are thus able to modify the SCCS files only through the use of `delta` and, possibly, `rmDEL` and `cdc`.

### C. A Basic Program

When a UNIX program is executed, the program is passed as argument 0, which is the name that invoked the program, and followed by any additional user-supplied arguments. Thus, if a program is given a number of links (names), the program may alter its processing depending upon which link invokes the program. This mechanism is used by an SCCS interface program to determine which SCCS command it should subsequently invoke [see `exec(2)` in the UNIX System User's Manual].

A generic interface program (`inter.c`, written in C language) is shown in Table 4.B. Note the reference to the (unsupplied) function "filearg". This is intended to demonstrate that the interface program may also be used as a preprocessor to SCCS commands. For example, function "filearg" could be used to modify file arguments to be passed to the SCCS command by supplying the full pathname of a file, thus avoiding extraneous typing by the user. Also, the program could supply any additional (default) keyletter arguments desired.

### D. Linking and Use

In general, the following demonstrates the steps to be performed by the SCCS administrator to create the SCCS interface program. It is assumed, for the purposes of the discussion, that the interface program `inter.c` resides in directory `/x1/xyz/scs`. Thus, the command sequence

```
cd /x1/xyz/scs
cc ... inter.c -o inter ...
```

compiles `inter.c` to produce the executable module `inter` (the "..." represent other arguments that may be required). The proper mode and the "set user ID on execution" bit are set by executing:

```
chmod 4755 inter
```

For example, new links are created by:

```
ln inter get
ln inter delta
ln inter rmDEL
```

The names of the links may be arbitrary provided the interface program is able to determine from them the names of SCCS commands to be invoked. Subsequently, any user whose shell parameter `PATH` [see `sh(1)` in the UNIX System User's Manual] specifies directory `/x1/xyz/scs` as the one to be searched first for executable commands may execute, e.g.:

```
get -e /x1/xyz/scs/s.abc
```

from any directory to invoke the interface program (via its link "get"). The interface program then executes `/usr/bin/get` (the actual SCCS `get` command) upon the named file. As previously mentioned, the interface program could be used to supply the pathname `/x1/xyz/scs` so that the user would only have to specify

```
get -e s.abc
```

to achieve the same results.



TABLE 4.A

## DETERMINATION OF NEW SID

| CASE | SID SPECIFIED* | -b KEYLETTER USED† | OTHER CONDITIONS                                  | SID RETRIEVED | SID OF DELTA TO BE CREATED |
|------|----------------|--------------------|---------------------------------------------------|---------------|----------------------------|
| 1    | none‡          | no                 | R defaults to mR                                  | mR.mL         | mR.(mL + 1)                |
| 2    | none‡          | yes                | R defaults to mR                                  | mR.mL         | mR.mL.(mB + 1).1           |
| 3    | R              | no                 | R > mR                                            | mR.mL         | R.1§                       |
| 4    | R              | no                 | R = mR                                            | mR.mL         | mR.(mL + 1)                |
| 5    | R              | yes                | R > mR                                            | mR.mL         | mR.mL.(mB + 1).1           |
| 6    | R              | yes                | R = mR                                            | mR.mL         | mR.mL.(mB + 1).1           |
| 7    | R              | —                  | R < mR and<br>R does not exist                    | hR.mL**       | hR.mL.(mB + 1).1           |
| 8    | R              | —                  | Trunk successor<br>in release > R<br>and R exists | R.mL          | R.mL.(mB + 1).1            |
| 9    | R.L            | no                 | No trunk successor                                | R.L           | R.(L + 1)                  |
| 10   | R.L            | yes                | No trunk successor                                | R.L           | R.L.(mB + 1).1             |
| 11   | R.L            | —                  | Trunk successor<br>in release ≥ R                 | R.L           | R.L.(mB + 1).1             |
| 12   | R.L.B          | no                 | No branch successor                               | R.L.B.mS      | R.L.B.(mS + 1)             |
| 13   | R.L.B          | yes                | No branch successor                               | R.L.B.mS      | R.L.(mB + 1).1             |
| 14   | R.L.B.S        | no                 | No branch successor                               | R.L.B.S       | R.L.B.(S + 1)              |
| 15   | R.L.B.S        | yes                | No branch successor                               | R.L.B.S       | R.L.(mB + 1).1             |
| 16   | R.L.B.S        | —                  | Branch successor                                  | R.L.B.S       | R.L.(mB + 1).1             |

\* "R", "L", "B", and "S" are the "release", "level", "branch", and "sequence" components of the SID, respectively; "m" means "maximum". Thus, for example, "R.mL" means "the maximum level number within release R"; "R.L.(mB + 1).1" means "the first sequence number on the new branch (i.e., maximum branch number plus 1) of level L within release R". Note that if the SID specified is of the form "R.L", "R.L.B", or "R.L.B.S", each of the specified components must exist.

† The -b keyletter is effective only if the b flag [see admin(1)] is present in the file. In this table, an entry of "—" means "irrelevant".

‡ This case applies if the d (default SID) flag is not present in the file. If the d flag is present in the file, the SID obtained from the d flag is interrupted as if it had been specified on the command line. Thus, one of the other cases in this table applies.

§ This case is used to force the creation of the first delta in a new release.

\*\* "hR" is the highest existing release that is lower than the specified, nonexistent, release R.



TABLE 4.8

## SCCS INTERFACE PROGRAM "inter.c"

```
main(argc, argv)
int argc;
char *argv[];
{
    register int i;
    char cmdstr[LENGTH]

    /*
    Process file arguments (those that don't begin with "-").
    */
    for (i = 1; i < argc; i++)
        if (argv[i][0] != '-')
            argv[i] = filearg(argv[i]);

    /*
    Get "simple name" of name used to invoke this program
    (i.e., strip off directory-name prefix, if any).
    */
    argv[0] = sname(argv[0]);

    /*
    Invoke actual SCCS command, passing arguments.
    */
    sprintf(cmdstr, "/usr/bin/%s", argv[0]);
    execv(cmdstr, argv);
}
```



NOTES

[Faint, illegible text within a large rectangular box, likely a placeholder for notes or a diagram.]



## 5. THE M4 MACRO PROCESSOR

### GENERAL

The M4 macro processor is a front end for rational Fortran (Ratfor) and the C programming languages. The "#define" statement in C language and the analogous "define" in Ratfor are examples of the basic facility provided by any macro processor.

At the beginning of a program, a symbolic name or symbolic constant can be defined as a particular string of characters. The compiler will then replace later unquoted occurrences of the symbolic name with the corresponding string. Besides the straightforward replacement of one string of text by another, the M4 macro processor provides the following features:

- arguments
- arithmetic capabilities
- file manipulation
- conditional macro expansion
- string and substring functions.

The basic operation of M4 is to read every alphanumeric token (string of letters and digits) input and determine if the token is the name of a macro. The name of the macro is replaced by its defining text, and the resulting string is pushed back onto the input to be rescanned. Macros may be called with arguments, in which case the arguments are collected and substituted into the right places in the defining text before the defining text is rescanned.

A list of 21 built-in macros provided by the M4 macro processor can be found in Table 5.A. The user also has the capability to define new macros. Built-ins and user-defined macros work exactly the same way except that some of the built-in macros have side effects on the state of the process.

To use the M4 macro processor, input the following command:

```
m4 [optional files]
```

Each argument file is processed in order. If there are no arguments or if an argument is "-", the standard input is read at that point. The processed text is written on the standard output which may be captured for subsequent processing with the following input:

```
m4 [files] >outputfile
```

### DEFINING MACROS

The primary built-in function of M4 is **define**, which is used to define new macros. The following input

```
define(name, stuff)
```

causes the string *name* to be defined as *stuff*. All subsequent occurrences of *name* will be replaced by *stuff*. *Name* must be alphanumeric and must begin with a letter (the underscore counts as a letter). *Stuff* is any text that contains balanced parentheses. Use of a slash may stretch *stuff* over multiple lines. Thus, as a typical example:

```
define(N, 100)
...
if (i > N)
```



defines *N* to be 100 and uses the symbolic constant *N* in a later if statement.

The left parenthesis must immediately follow the word **define** to signal that **define** has arguments. If a user-defined macro or built-in name is not followed immediately by "(", it is assumed to have no arguments. Macro calls have the following general form:

```
name(arg1,arg2,...argn)
```

A macro name is only recognized as such if it appears surrounded by nonalphanumerics. Using the following example:

```
define(N, 100)
...
if (NNN > 100)
```

the variable *NNN* is absolutely unrelated to the defined macro *N* even though the variable contains a lot of *N*s.

Macros may be defined in terms of other names. For example,

```
define(N, 100)
define(M, N)
```

defines both *M* and *N* to be 100. If *N* is redefined and subsequently changes, *M* retains the value of 100 not *N*.

The M4 macro processor expands macro names into their defining text as soon as possible. The string *N* is immediately replaced by 100. Then the string *M* is also immediately replaced by 100. The overall result is the same as using the following input in the first place:

```
define(M, 100)
```

The order of the definitions can be interchanged as follows:

```
define(M, N)
define(N, 100)
```

Now *M* is defined to be the string *N*, so when the value of *M* is requested later, the result is the value of *N* at that time (because the *M* will be replaced by *N* which will be replaced by 100).

The more general solution is to delay the expansion of the arguments of **define** by quoting them. Any text surrounded by left and right single quotes is not expanded immediately but has the quotes stripped off. The value of a quoted string is the string stripped of the quotes. If the input is

```
define(N, 100)
define(M, 'N')
```

the quotes around the *N* are stripped off as the argument is being collected. The results of using quotes is to define *M* as the string *N*, not 100. The general rule is that M4 always strips off one level of single quotes whenever it evaluates something. This is true even outside of macros. If the word **define** is to appear in the output, the word must be quoted in the input as follows:

```
'define' = 1;
```



Another example of using quotes is redefining *N*. To redefine *N*, the evaluation must be delayed by quoting:

```
define(N, 100)
...
define('N', 200)
```

In M4, it is often wise to quote the first argument of a macro. The following example will not redefine *N*:

```
define(N, 100)
...
define(N, 200)
```

The *N* in the second definition is replaced by 100. The result is equivalent to the following statement:

```
define(100, 200)
```

This statement is ignored by M4 since only things that look like names can be defined.

If left and right single quotes are not convenient for some reason, the quote characters can be changed with the following built-in macro:

```
changequote([, ])
```

The built-in **changequote** makes the new quote characters the left and right brackets. The original characters can be restored by using **changequote** without arguments as follows:

```
changequote
```

There are two additional built-ins related to **define**. The **undefine** macro removes the definition of some macro or built-in as follows:

```
undefine('N')
```

The macro removes the definition of *N*. Built-ins can be removed with **undefine**, as follows:

```
undefine('define')
```

But once removed, the definition can not be reused.

The built-in **ifdef** provides a way to determine if a macro is currently defined. In particular, M4 has predefined the names *pdp11* and *u3b* on the corresponding systems. Depending on the system, a definition appropriate for the particular machine can be made as follows:

```
ifdef('pdp11', 'define(wordsize,16)')
ifdef('u3b', 'define(wordsize,32)')
```

Remember to use the quotes.

The **ifdef** macro actually permits three arguments. If the first argument is defined, the value of **ifdef** is the second argument, otherwise the third. If there is no third argument, the value of **ifdef** is null. If the name is undefined, the value of **ifdef** is then the third argument, as in:

```
ifdef('unix', on UNIX, not on UNIX)
```



## ARGUMENTS

So far the simplest form of macro processing has been discussed which is replacing one string by another (fixed) string. User-defined macros may also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its **define**), any occurrence of **\$n** will be replaced by the *n*th argument when the macro is actually used. Thus, the macro **bump** defined as

```
define(bump, $1 = $1 + 1)
```

generates code to increment its argument by 1. The 'bump(x)' statement is equivalent to 'x = x + 1.'

A macro can have as many arguments as needed, but only the first nine are accessible (**\$1** through **\$9**). The macro name is **\$0** although that is less commonly used. Arguments that are not supplied are replaced by null strings, so a macro can be defined which simply concatenates its arguments like this:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Thus, 'cat(x, y, z)' is equivalent to 'xyz'. Arguments **\$4** through **\$9** are null since no corresponding arguments were provided. Leading unquoted blanks, tabs, or new lines that occur during argument collection are discarded. All other white space is retained. Thus:

```
define(a, b c)
```

defines 'a' to be 'b c'.

Arguments are separated by commas, but parentheses are counted properly so a comma protected by parentheses does not terminate an argument. For example:

```
define(a, (b,c))
```

has only two arguments. The first argument is **a**. The second is literally **(b,c)**. A bare comma or parenthesis can be inserted by quoting it.

## ARITHMETIC BUILT-INS

The M4 provides three built-in functions for doing arithmetic on integers (only). The simplest is **incr** which increments its numeric argument by 1. The built-in **decr** decrements by 1. Thus to handle the common programming situation where a variable is to be defined as "one more than *N*", use the following:

```
define(N, 100)
define(N1, 'incr(N))
```

Then *N1* is defined as one more than the current value of *N*.

The more general mechanism for arithmetic is a built in called **eval** which is capable of arbitrary arithmetic on integers. The operators in decreasing order of precedence are:

```
unary + and -
** or ^ (exponentiation)
* / % (modulus)
+ -
== != < <= > >=
! (not)
& or && (logical and)
| or # (logical or).
```



Parentheses may be used to group operations where needed. All the operands of an expression given to **eval** must ultimately be numeric. The numeric value of a true relation (like  $1 > 0$ ) is 1, and false is 0. The precision in **eval** is 32 bits under the UNIX operating system.

As a simple example, define *M* to be " $2 == N + 1$ " using **eval** as follows:

```
define(N, 3)
define(M, 'eval(2==N+1)')
```

The defining text for a macro should be quoted unless the text is very simple. Quoting the defining text usually gives the desired result and is a good habit to get into.

### FILE MANIPULATION

A new file can be included in the input at any time by the built-in function **include**. For example:

```
include(filename)
```

inserts the contents of *filename* in place of the **include** command. The contents of the file is often a set of definitions. The value of **include** (**include**'s replacement text) is the contents of the file. If needed, the contents can be captured in definitions, etc.

A fatal error occurs if the file named in **include** cannot be accessed. To get some control over this situation, the alternate form **sinclude** can be used. The built-in **sinclude** (silent include) says nothing and continues if the file named can not be accessed.

The output of M4 can be diverted to temporary files during processing, and the collected material can be output upon command. The M4 maintains nine of these diversions, numbered 1 through 9. If the built-in macro

```
divert(n)
```

is used, all subsequent output is put onto the end of a temporary file referred to as *n*. Diverting to this file is stopped by the **divert** or **divert(0)** command which resumes the normal output process.

Diverted text is normally output all at once at the end of processing with the diversions output in numerical order. Diversions can be brought back at any time by appending the new diversion to the current diversion. Output diverted to a stream other than 0 through 9 is discarded. The built-in **undivert** brings back all diversions in numerical order. The built-in **undivert** with arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted text as does diverting into a diversion whose number is not between 0 and 9, inclusive.

The value of **undivert** is *not* the diverted text. Furthermore, the diverted material is *not* rescanned for macros. The built-in **divnum** returns the number of the currently active diversion. The current output stream is zero during normal processing.

### SYSTEM COMMAND

Any program in the local operating system can be run by using the **syscmd** built-in. For example:

```
syscmd(date)
```

on the UNIX system runs the **date** command. Normally, **syscmd** would be used to create a file for a subsequent **include**.



To facilitate making unique file names, the built-in **maketemp** is provided with specifications identical to the system function *mktemp*. The **maketemp** macro fills in a string of XXXXX in the argument with the process id of the current process.

## CONDITIONALS

Arbitrary conditional testing is performed via built-in **ifelse**. In the simplest form

```
ifelse(a, b, c, d)
```

compares the two strings *a* and *b*. If *a* and *b* are identical, **ifelse** returns the string *c*. Otherwise, string *d* is returned. Thus, a macro called **compare** can be defined as one which compares two strings and returns "yes" or "no" if they are the same or different as follows:

```
define(compare, 'ifelse($1, $2, yes, no)')
```

Note the quotes which prevents evaluation of **ifelse** occurring too early. If the fourth argument is missing, it is treated as empty.

The built-in **ifelse** can actually have any number of arguments and provides a limited form of multiway decision capability. In the input:

```
ifelse(a, b, c, d, e, f, g)
```

if the string *a* matches the string *b*, the result is *c*. Otherwise, if *d* is the same as *e*, the result is *f*. Otherwise, the result is *g*. If the final argument is omitted, the result is null, so

```
ifelse(a, b, c)
```

is *c* if *a* matches *b*, and null otherwise.

## STRING MANIPULATION

The built-in **len** returns the length of the string (number of characters) that makes up its argument. Thus:

```
len(abcdef)
```

is 6, and **len((a,b))** is 5.

The built-in **substr** can be used to produce substrings of strings. Using input, **substr(s, i, n)** returns the substring of *s* that starts at the *i*th position (origin zero) and is *n* characters long. If *n* is omitted, the rest of the string is returned. Inputting

```
substr('now is the time',1)
```

returns the following string:

```
ow is the time.
```

If *i* or *n* are out of range, various actions occur.

The built-in **index(s1, s2)** returns the index (position) in *s1* where the string *s2* occurs or -1 if it doesn't occur. As with **substr**, the origin for strings is 0.



The built-in **translit** performs character transliteration and has the general form

```
translit(s, f, t)
```

which modifies *s* by replacing any character found in *f* by the corresponding character of *t*. Using input

```
translit(s, aeiou, 12345)
```

replaces the vowels by the corresponding digits. If *t* is shorter than *f*, characters which do not have an entry in *t* are deleted. As a limiting case, if *t* is not present at all, characters from *f* are deleted from *s*. So

```
translit(s, aeiou)
```

would delete vowels from *s*.

There is also a built-in called **dnl** which deletes all characters that follow it up to and including the next new line. The **dnl** macro is useful mainly for throwing away empty lines that otherwise tend to clutter up M4 output. Using input

```
define(N, 100)  
define(M, 200)  
define(L, 300)
```

results in a new line at the end of each line that is not part of the definition. So the new line is copied into the output where it may not be wanted. If the built-in **dnl** is added to each of these lines, the new lines will disappear. Another method of achieving the same results is to input

```
divert(-1)  
define(...)  
...  
divert.
```

## PRINTING

The built-in **errprint** writes its arguments out on the standard error file. An example would be:

```
errprint ('fatal error')
```

The built-in **dumpdef** is a debugging aid which dumps the current names and definitions of items named as arguments. If no arguments are given, then all current names and definitions are printed. Do not forget to quote the names.



TABLE 5.A  
BUILT-IN MACROS

| MACRO NAME  | FUNCTION                                                                                                            |
|-------------|---------------------------------------------------------------------------------------------------------------------|
| changequote | Restores original characters or makes new quote characters the left and right brackets                              |
| changecom   | Changes left and right comment markers from the default # and new line                                              |
| decr        | Returns the value of its argument decremented by 1                                                                  |
| define      | Defines new macros                                                                                                  |
| defn        | Returns the quoted definition of its argument(s)                                                                    |
| divert      | Diverts output to one-of-ten diversions                                                                             |
| divnum      | Returns the number of the currently active diversion                                                                |
| dnl         | Reads and discards characters up to and including the next new line                                                 |
| dumpdef     | Dumps the current names and definitions of items named as arguments                                                 |
| errprint    | Prints its arguments on the standard error file                                                                     |
| eval        | Performs arbitrary arithmetic on integers                                                                           |
| ifdef       | Determines if a macro is currently defined                                                                          |
| ifelse      | Performs arbitrary conditional testing                                                                              |
| include     | Returns the contents of the file named in the argument. A fatal error occurs if the file named can not be accessed. |
| incr        | Returns the value of its argument incremented by 1                                                                  |
| index       | Returns the position where the second argument begins in the first argument of index                                |
| len         | Returns the number of characters that makes up its argument                                                         |
| m4exit      | Causes immediate exit from M4                                                                                       |
| m4wrap      | Pushes the exit code back at final EOF                                                                              |
| maketemp    | Facilitates making unique file names                                                                                |
| popdef      | Removes current definition of its argument(s) exposing any previous definition                                      |



TABLE 5.A (Contd)

## BUILT-IN MACROS

| MACRO NAME | FUNCTION                                                                                                                     |
|------------|------------------------------------------------------------------------------------------------------------------------------|
| pushdef    | Defines new macros, but saves any previous definition                                                                        |
| shift      | Returns all arguments of shift except the first argument                                                                     |
| sinclude   | Returns the contents of the file named in the arguments. The macro remains silent and continues if the file is inaccessible. |
| substr     | Produces substrings of strings                                                                                               |
| syscmd     | Executes the UNIX system command given in the first argument                                                                 |
| traceoff   | Turns macro trace off                                                                                                        |
| traceon    | Turns the macro trace on                                                                                                     |
| translit   | Performs character transliteration                                                                                           |
| undefine   | Removes user-defined or built-in macro definitions                                                                           |
| undivert   | Discards the diverted text                                                                                                   |







## 6. THE "awk" PROGRAMMING LANGUAGE

### GENERAL

The **awk** programming language is designed to scan a set of files for lines that match any of a set of patterns which the user has specified. For each pattern, an action can be specified. The specified action will be performed on each line or fields of lines that match the pattern. The **awk** language is designed to make many common information retrieval and text manipulation tasks easy to state and to perform.

Readers familiar with the program **grep** will recognize the approach, although in **awk** the patterns may be more general than in **grep**, and the actions allowed are more involved than printing the matching line. For example, the **awk** program

```
{print $3, $2}
```

prints the third and second columns of a table in that order. The program

```
$2 ~ /A|B|C/
```

prints all input lines with an A, B, or C in the second field. The program

```
$1 != prev { print; prev = $1 }
```

prints all lines in which the first field is different from the previous first field.

#### A. Usage

The command

```
awk program [files]
```

scans each input file, or on the standard input if there are no files, for lines that match any of a set of patterns specified in *program*. With each pattern in *program*, there may be an associated action that will be performed when a line of the input matches the pattern. The **awk** commands may appear literally in *program*, or the statements can also be placed in a file *pfile* and executed by the command:

```
awk -f pfile [files]
```

#### B. Program Structure

An **awk** program is a sequence of statements of the form:

```
pattern          { action }
pattern          { action }
...
```

Each line of input is matched against each of the *patterns* in succession. For each pattern that matches, the associated *action* is executed. When all the patterns have been tested, the next line of input is fetched and the matching process starts over.

Either the pattern or the action may be omitted, but not both. If there is no action for a pattern, the matching line is simply copied to the output. (A line which matches several patterns can be printed several times.) If there is no pattern for an action, then the action is performed for every input line. A line which does not match a pattern is ignored.



Since patterns or actions may be omitted, actions must be enclosed in braces to distinguish them from patterns in the program.

The `awk` patterns may include arbitrary boolean combinations of regular expressions and of relational operators on strings, numbers, fields, variables, and array elements. Actions may include the same pattern-matching constructions as in patterns, as well as arithmetic and string expressions and assignments, `if-else`, `while`, `for` statements, and multiple output streams.

### C. Records and Fields

The variable `FILENAME` contains the name of the current input file. The input to `awk` is divided into "records" terminated by a record separator. The default record separator is a newline. However, the input record separator may be changed; so by default, `awk` processes its input a line at a time. The number of the current record is available in a variable named `NR`.

Each input record is considered to be divided into "fields". The default field separator is white space (blanks or tabs); however, the input field separator may be changed. Fields are referred to as `$1`, `$2`, etc. where `$1` is the first field, and `$0` is the entire input record. Fields may be assigned to a numeric or string value. The number of fields in the current record is available in the variable `NF`.

The variables `FS` and `RS` refer to the input field and record separators, respectively; they may be changed at any time to any single character. The optional command-line argument `-Fc` may also be used to set `FS` to the character `c`.

If the record separator is empty, an empty input line is taken as the record separator; and blanks, tabs, and newlines are treated as field separators.

### D. Printing

When there is no pattern for an action, the action is executed for all lines. The simplest action is to print some or all of a record; this is accomplished by the `awk` command `print`. The `awk` program

```
{ print }
```

prints each record copying the input to the output. A more useful program is to print a field or fields from each record. For instance,

```
print $2, $1
```

prints the first two fields in reverse order. Items separated by a comma in the print statement will be separated by the current output field separator, referenced by the variable `OFS`, when output. Items not separated by commas will be concatenated, so

```
print $1 $2
```

prints the first and second fields together.

The predefined variables `NF` and `NR` can be used in the print command; for example:

```
{ print NR, NF, $0 }
```

prints each record preceded by the record number and the number of fields in the record.



Output may be diverted to multiple files; the program

```
{ print $1 >"foo1" ; print $2>"foo2" }
```

writes the first field, \$1, on file *foo1*, and the second field on file *foo2*. The >> notation can also be used:

```
print $1 >> "foo"
```

appends the first field, \$1, to file *foo*. (In each case, the output files are created if necessary.) The file name can be a variable, a field, or a constant; for example:

```
print $1 >$2
```

uses the contents of field 2 as a file name.

There is a limit on the number of output files; currently the limit is 10.

Similarly, output can be piped into another process (on the UNIX operating system only); for instance:

```
print | "mail bwk"
```

mails the output to *bwk*.

The variables *OFS* and *ORS* may be used to change the current output field separator and output record separator. The default output field separator is a blank, and the default output record separator is a newline. The output record separator is appended to the output of the **print** statement.

The **awk** language also provides the **printf** statement for output formatting:

```
printf format expr, expr, ...
```

formats the expressions in the list according to the specification in *format* and prints them. The statement

```
printf " %8.2f %10ld\n" , $1, $2
```

prints the first field, \$1, as a floating point number eight digits wide with two after the decimal point and prints the second field, \$2, as a 10-digit long decimal number; followed by a new line. No output separators are produced automatically. In this example, the two fields will be separated by the current output field separator. The version of **printf** used in the **awk** programming language is identical to that used in the C programming language.

## PATTERNS

The "pattern" which precedes an "action" in an **awk** program acts as a selector to determine whether the action is to be executed. A variety of expressions may be used as patterns: regular expressions, arithmetic relational expressions, string-valued expressions, and arbitrary boolean combinations of these.

### A. "BEGIN" and "END"

The special pattern **BEGIN** matches the beginning of the input before the first record is read. The pattern **END** matches the end of the input after the last record has been processed. **BEGIN** and **END** provides a way to gain control of the program before and after processing.



As an example, using the variable *FS*, the field separator can be set to a colon by

```
BEGIN { FS = ":" }  
... rest of program ...
```

or, using the variable *NR*, the input lines may be counted by

```
END { print NR }
```

If **BEGIN** is present, it must be the first pattern; if **END** is present, it must be the last pattern in the program.

## B. Regular Expressions

The simplest regular expression is a literal string of characters enclosed in slashes, for example:

```
/smith/
```

This is a complete **awk** program that prints all lines which contain any occurrence of the name "smith". If a line contains "smith" as part of a larger word, it will also be printed, as in

```
blacksmithing
```

The **awk** regular expressions include the regular expression forms found in the UNIX text editor **ed** and **grep** (without back referencing). In addition, **awk** allows parentheses for grouping, **|** for alternatives, **+** for "one or more", and **?** for "zero or one", all as in the **lex** programming language. Character classes may be abbreviated as **{a-zA-Z0-9}** to represent the set of all letters and digits. As an example, the **awk** program

```
/[Aa]ho[Ww]einberger[Kk]ernighan/
```

prints all lines containing any of the names "Aho", "Weinberger", or "Kernighan", whether capitalized or not.

Regular expressions (with the extensions listed above) must be enclosed in slashes, just as in the programs **ed** and **sed**. Within a regular expression, blanks and the regular expression metacharacters are significant. To turn off the special meaning of one of the regular expression characters, precede it with a backslash. An example is the pattern:

```
/\/.*\/
```

which matches any string of characters enclosed in slashes.

Any field or variable can be specified to match (or not match) a regular expression with the operators **~** and **!~**. The program

```
$1 ~ /[jJ]ohn/
```

prints all lines where the first field matches "john" or "John". Notice that the program will also match "Johnson", "St. Johnsbury", etc. To restrict the program to match lines or fields that contain only "[jJ]ohn", use

```
$1 ~ /^[jJ]ohn$/
```

The caret **^** refers to the beginning of a line or field; the dollar sign **\$** refers to the end of a line or field.



### C. Relational Expressions

An **awk** pattern can be a relational expression involving the usual relational operators **<**, **<=**, **==**, **!=**, **>=**, and **>**. An example is

```
$2 > $1 + 100
```

which selects lines where the second field is at least 100 greater than the first field. Similarly,

```
NF % 2 == 0
```

prints lines that contain an even number of fields.

In relational tests if neither operand is numeric, a string comparison is made; otherwise, the operand is numeric. Thus:

```
$1 >= "s"
```

selects lines that begin with an "s", "t", "u", etc. In the absence of any other information, fields are treated as strings; therefore, the program

```
$1 > $2
```

will perform a string comparison.

### D. Combinations of Patterns

A pattern can be any boolean combination of patterns, using the operators **#** (or), **&&** (and), and **!** (not). For example:

```
$1 >= "s" && $1 < "t" && $1 != "smith"
```

selects lines where the first field begins with "s" but is not "smith". The **&&** and **#** guarantee that the operands will be evaluated from left to right; evaluation stops when the truth or falsehood is determined.

### E. Pattern Ranges

A "pattern" may consist of two patterns separated by a comma; in this case, the action is performed for all lines between an occurrence of the first pattern and the next occurrence of the second. For instance,

```
pat1, pat2 { ... }
```

will perform the action for each line between an occurrence of "pat1" and the next occurrence of "pat2" (inclusive). An example is:

```
/start/, /stop/
```

which prints all lines between the patterns "start" and "stop", while:

```
NR == 100, NR == 200 { ... }
```

performs the action for lines 100 through 200 of the input.



## ACTIONS

An **awk** action is a sequence of action statements terminated by newlines or semicolons. These action statements can be used for a variety of bookkeeping and string manipulating tasks.

## A. Built-in Functions

The **awk** language provides a "length" function to compute the length of a string of characters. The following program prints each record preceded by its length:

```
{print length, $0}
```

The function **length** is a "pseudo-variable" which yields the length of the current record; **length(argument)** is a function which yields the length of its argument. The argument may be any expression. The following is equivalent to the previous program:

```
{print length($0), $0}
```

Also provided by **awk** are the arithmetic functions **sqrt**, **log**, **exp**, and **int** for square root, base e logarithm, exponential, and integer part of their respective arguments.

The name of one of these built-in functions, without argument or parentheses, stands for the value of the function on the entire record. The program

```
length < 10 # length > 20
```

prints lines whose length is less than 10 or greater than 20.

The function **fBbstr(s, m, n)** produces the substring of "s" that begins at position "m" (origin 1) and is at most "n" characters long. If "n" is omitted, the substring goes to the end of "s". The function **index(s1, s2)** returns the position where the string "s2" occurs in "s1" or zero if it does not.

The function **sprintf(f, e1, e2, ...)** produces the value of the expressions "e1", "e2", etc., in the **printf** format specified by "f". For example

```
x = sprintf(" %8.2f %10ld ", $1, $2)
```

sets **x** to the string produced by formatting the values of the first field, **\$1**, and the second field, **\$2**.

## B. Variables, Expressions, and Assignments

The **awk** variables take on numeric (floating point) or string values according to context. For example, in:

```
x = 1
```

**x** is clearly a number, while in:

```
x = " smith "
```

**x** is clearly a string. Strings are converted to numbers and vice versa whenever context demands it. For instance:

```
x = " 3 " + " 4 "
```

assigns 7 to **x**. Strings which cannot be interpreted as numbers in a numerical context will generally have numeric value zero. To force an expression to be treated as a number, add 0 (zero) to it; to force an expression to be treated as a string, concatenate the null string ( " " ) to it.



By default, variables (other than built-ins) are initialized to the null string, which has numerical value zero; this eliminates the need for most **BEGIN** sections. For example, the sums of the first two fields can be computed by:

```

                                { s1 += $1; s2 += $2 }
END                            { print s1, s2 }
```

Arithmetic is done internally in floating point. The arithmetic operators are +, -, \*, /, and % (mod). The C language increment ++ and decrement -- operators are available; also the assignment operators +=, -=, \*=, /=, and %= are available. These operators may all be used in expressions.

### C. Field Variables

Fields in **awk** share essentially all of the properties of variables—they may be used in arithmetic or string operations and may be assigned to a numeric or string value. One can replace the first field with a sequence number with:

```
{ $1 = NR; print }
```

or accumulate two fields into a third:

```
{ $1 = $2 + $3; print $0 }
```

or assign a string to a field:

```

{ if ($3 > 1000)
  $3 = "too big"
  print
}
```

which replaces the third field with "too big" when the third field is greater than 1000 and prints the record.

Field references may be numerical expressions, as in:

```
{ print $i, $(i+1), $(i+n) }
```

Whether a field is numeric or string depends on context; in ambiguous cases like

```
if ($1 == $2) ...
```

fields are treated as strings.

Each input line is split into fields automatically as necessary. It is also possible to split any variable or string into fields;

```
n = split(s, array, sep)
```

splits the string "s" into "array[1], ..., array[n]". The number of elements found is returned. If the "sep" argument is provided, it is used as the field separator; otherwise, the field separator is that which is referenced by the variable *FS*.



## D. String Concatenation

Strings may be concatenated. For example:

```
length($1 $2 $3)
```

returns the length of the first three fields; or in a **print** statement,

```
print $1 " is " $2
```

prints the two fields separated by "is". Variables and numeric expressions may also appear in concatenations.

## E. Arrays

Array elements are not declared; the elements are initialized when mentioned. Subscripts may have any non-null value including non-numeric strings. As an example of a conventional numeric subscript, the statement

```
x[NR] = $0
```

assigns the current input record to the *NR*-th element of the array *x*. In fact, it is possible in principle (though perhaps slow) to process the entire input in a random order with the **awk** program

```
END { x[NR] = $0 }
{ ... program ... }
```

The first action records each input line in the array *x*.

Array elements may be named by non-numeric values, which gives **awk** a capability rather like the associative memory of Snobol tables. Suppose the input contains fields with values like "apple", "orange", etc. Then the program

```
/apple/ { x["apple"]++ }
/orange/ { x["orange"]++ }
END { print x["apple"], x["orange"] }
```

increments counts for the named array elements and prints them at the end of the input.

Any expression can be used as a subscript in an array reference. Thus:

```
x[$1] = $2
```

uses the first field of a record (as a string) to index the array *x*.

Suppose each line of input contains two fields — a name and a nonzero value. Names may be repeated; the task is to print a list of each unique name followed by the sum of all the values for that name. This can be done with the program

```
END { amount[$1] += $2 }
{ for (name in amount)
  print name, amount[name] }
```

To sort the output, replace the last line by

```
print name, amount[name] | "sort "
```



## F. Flow-of-Control Statements

The **awk** language also provides the basic flow-of-control statements **if-else**, **while**, **for**, and statement grouping with braces, as in the C programming language. The **if-else** statement is exactly like that of the C language and was previously shown in the subpart "Field Variables". The condition in parentheses of an **if-else** statement is evaluated; if it is true, the statement following the **if** is performed. The **else** part is optional.

The **while** statement is exactly like that of the C language. For example, to print all input fields one per line:

```
i = 1
while (i <= NF) {
    print $i
    ++i
}
```

The **for** statement is also like that of the C language;

```
for (i = 1; i <= NF; i++)
    print $i
```

performs the same task as the **while** statement above.

There is an alternate form of the **for** statement which is suited for accessing the elements of an associative array:

```
for (i in array)
    statement
```

does statement with *i* set in turn to each element of **array**. The elements are accessed in an apparently random order. Confusion will develop if *i* is altered or if any new elements are accessed during the loop.

The expression in the condition part of an **if**, **while**, or **for** can include relational operators like **<**, **<=**, **>**, **>=**, **==** (equal to), and **!=** (not equal to); regular expression matches with the match operators **~** and **!~**; the logical operators **||**, **&&**, and **!**; and parentheses for grouping.

The **break** statement causes an immediate exit from an enclosing **while** or **for** loop; the **continue** statement causes the next iteration of the enclosing loop to begin.

The statement **next** causes the **awk** program to skip immediately to the next record and begin scanning the patterns from the top. The statement **exit** causes the program to behave as if the end of the input had occurred.

Comments may be placed in **awk** programs, but they must begin with the character **#** and terminate with the end of the line, for example:

```
print x, y    # this is a comment
```



**NOTES**



## 7. ARBITRARY PRECISION DESK CALCULATOR LANGUAGE (BC)

### GENERAL

The arbitrary precision desk calculator language (BC) is a language and compiler for doing arbitrary precision arithmetic under the UNIX operating system. The output of the compiler is interpreted and executed by a collection of routines which can input, output, and do arithmetic on infinitely large integers and on scaled fixed-point numbers. These routines are based on a dynamic storage allocator. Overflow does not occur until all available core storage is exhausted.

The BC language has a complete control structure as well as immediate-mode operation. Functions can be defined and saved for later execution. A small collection of library functions is also available, including sin, cos, arctan, log, exponential, and Bessel functions of integer order.

The BC compiler was written to make conveniently available a collection of routines (called DC) which are capable of doing arithmetic on integers of arbitrary size. The compiler is not intended to provide a complete programming language. It is a minimal language facility.

Some of the uses of this compiler are

- to do computation with large integers
- to do computation accurate to many decimal places
- conversion of numbers from one base to another base.

There is a scaling provision that permits the use of decimal point notation. Provision is also made for input and output in bases other than decimal. Numbers can be converted from decimal to octal by simply setting the output base to equal eight.

The actual limit on the number of digits that can be handled depends on the amount of core storage available. Manipulation of numbers with many hundreds of digits is possible even on the smallest versions of the UNIX operating system.

The syntax of BC is very similar to that of the C language. This enables users who are familiar with C language to easily work with BC.

### SIMPLE COMPUTATIONS WITH INTEGERS

The simplest kind of statement is an arithmetic expression on a line by itself. For instance, if you type in the addition of two numbers (with the + operator) such as

```
142857 + 285714
```

the program responds immediately with the sum

```
428571.
```

The operators -, \*, /, %, and ^ can also be used. They indicate subtraction, multiplication, division, remainder, and exponentiation, respectively. Division of integers produces an integer result truncated toward zero. Division by zero produces an error comment.

Any term in an expression may be prefixed by a minus sign to indicate that it is to be negated (the unary minus sign). The expression

```
7+-3
```



is interpreted to mean that  $-3$  is to be added to  $7$ .

More complex expressions with several operators and with parentheses are interpreted just as in Fortran, with  $^$  having the greatest binding power, then  $*$ ,  $\%$ , and  $/$ , and finally,  $+$  and  $-$ . Contents of parentheses are evaluated before material outside the parentheses. Exponentiations are performed from right to left and the other operators from left to right. The two expressions

$a^b^c$  and  $a^{(b^c)}$

are equivalent as are the two expressions

$a*b*c$  and  $(a*b)*c$ .

However, BC shares with Fortran and C language the undesirable convention that

$a/b*c$  is equivalent to  $(a/b)*c$ .

Internal storage registers to hold numbers have single lowercase letter names. The value of an expression can be assigned to a register in the usual way. The statement

$x = x + 3$

has the effect of increasing by three the value of the contents of the register named  $x$ . When, as in this case, the outermost operator is an "=", the assignment is performed; but the result is not printed. Only 26 of these named storage registers are available.

There is a built-in square root function whose result is truncated to an integer (see the part on "SCALING"). Entering the lines

$x = \text{sqrt}(191)$   
 $x$

produces the printed result

13.

## BASES

There are two special internal quantities; **ibase** (input base) and **obase** (output base). The contents of **ibase**, initially set to 10 (decimal), determines the base used for interpreting numbers read in. For example, the input lines

$\text{ibase} = 8$   
11

will produce the output line

9

and the system is ready to do octal to decimal conversions. Beware, however, of trying to change the input base back to decimal by typing

$\text{ibase} = 10$



Because the number 10 is interpreted as octal, this statement will have no effect. For dealing in hexadecimal notation, the characters A through F are permitted in numbers (regardless of what base is in effect) and are interpreted as digits having values 10 through 15, respectively. The statement

```
ibase = A
```

will change the base to decimal regardless of what the current input base is. Negative and large positive input bases are permitted but are useless. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 and greater than 16.

The content of **obase**, initially 10 (decimal), is used as the base for output numbers. The input lines

```
obase = 16  
1000
```

will produce the output line

```
3E8
```

which is to be interpreted as a 3-digit hexadecimal number. Very large output bases are permitted and are sometimes useful. For example, large numbers can be output in groups of five digits by setting **obase** to 100000. Strange output bases (i.e., 1, 0, or negative) are handled appropriately.

Very large numbers are split across lines with 70 characters per line. Lines which are continued end with a backslash (\). Decimal output conversion is practically instantaneous, but output of very large numbers (i.e., more than 100 digits) with other bases is rather slow. Nondecimal output conversion of a 100-digit number takes about 3 seconds.

The **ibase** and **obase** have no effect on the course of internal computation or on the evaluation of expressions. They only affect input and output conversions, respectively.

## SCALING

A third special internal quantity called **scale** is used to determine the scale of calculated quantities. The number of digits after the decimal point of a number is referred to as its scale. Numbers may have up to 99 decimal digits after the decimal point. This fractional part is retained in further computations.

The contents of **scale** must be no greater than 99 and no less than 0. It is initially set to 0. However, appropriate scaling can be arranged when more than 99 fraction digits are required.

When two scaled numbers are combined by means of one of the arithmetic operations, the result has a scale determined by the following rules:

- Addition and subtraction—The scale of the result is the larger of the scales of the two operands. In this case, there is never any truncation of the result.
- Multiplication—The scale of the result is never less than the maximum of the two scales of the operands and never more than the sum of the scales of the operands. Subject to those two restrictions, the scale of the result is set equal to the contents of the internal quantity **scale**.
- Division—The scale of a quotient is the contents of the internal quantity **scale**. The scale of a remainder is the sum of the scales of the quotient and the divisor.
- Exponentiation—The result of an exponentiation is scaled as if the implied multiplications were performed. An exponent must be an integer.



- Square root—The scale of a square root is set to the maximum of the scale of the argument and the contents of **scale**.

All of the internal operations are actually carried out in terms of integers with digits being discarded when necessary. In every case where digits are discarded, truncation and not rounding is performed.

The internal quantities **scale**, **ibase**, and **obase** can be used in expressions just like other variables. The input line

```
scale = scale + 1
```

increases the value of **scale** by one, and the input line

```
scale
```

causes the current value of **scale** to be printed.

The value of **scale** retains its meaning as a number of decimal digits to be retained in internal computation even when **ibase** or **obase** are not equal to 10. The internal computations (which are still conducted in decimal regardless of the bases) are performed to the specified number of decimal digits, never hexadecimal, octal, or any other kind of digits.

## FUNCTIONS

The name of a function is a single lowercase letter. Function names are permitted to coincide with simple variable names. Twenty-six different defined functions are permitted in addition to the 26 variable names. The input line

```
define a(x){
```

begins the definition of a function with one argument. This line must be followed by one or more statements which make up the body of the function ending with a right brace ( **}** ). The general form of a function is

```
define a(x) {  
    ...  
    return  
}
```

Return of control from a function occurs when a **return** statement is executed or when the end of the function is reached. The **return** statement can take either of the two forms:

```
return  
return(x)
```

In the first case, the value of the function is 0; and in the second, the value of the function is the expression in parentheses.

Variables used in the function can be declared as automatic by a statement of the form:

```
auto x,y,z
```

There can be only one **auto** statement in a function, and it must be the first statement in the definition. These automatic variables are allocated space and initialized to zero on entry to the function and thrown away on return (exit). The values of any variables with the same names outside the function are not disturbed. Functions



may be called recursively and the automatic variables at each level of call are protected. The parameters named in a function definition are treated in the same way as the automatic variables of that function with the single exception that they are given a value on entry to the function. An example of a function definition is

```
define a(x,y){  
    auto z  
    z = x*y  
    return(z)  
}
```

The value of this function `a`, when called, will be the product of its two arguments, "x" and "y".

A function is called by the appearance of its name followed by a string of arguments enclosed in parentheses and separated by commas. The result is unpredictable if the wrong number of arguments is used.

Functions with no arguments are defined and called using parentheses with nothing between them: `b()`.

If the function `a` above has been defined, then the line

```
a(7,3.14)
```

would cause the result 21.98 to be printed, and the line

```
z = a(a(3,4),5)
```

would cause the result 60 to be printed.

#### SUBSCRIPTED VARIABLES

A single lowercase letter variable name followed by an expression in brackets is called a subscripted variable (an array element). The variable name is called the array name, and the expression in brackets is called the subscript. Only 1-dimensional arrays are permitted. The names of arrays are permitted to coincide with the names of simple variables and function names. Any fractional part of a subscript is discarded before use. Subscripts must be greater than or equal to 0 and less than or equal to 2047.

Subscripted variables may be used in expressions, in function calls, and in return statements.

An array name may be used as an argument to a function or may be declared as automatic in a function definition by the use of empty brackets:

```
f(a[])  
define f(a[])  
auto a[]
```

When an array name is so used, the whole contents of the array are copied for the use of the function and thrown away on exit from the function. Array names that refer to whole arrays cannot be used in any other contexts.

#### CONTROL STATEMENTS

The `if`, `while`, and `for` statements may be used to alter the flow within programs or to cause iteration. The range of each of them is a statement or a compound statement consisting of a collection of statements enclosed in braces. They are written in the following way:

```
if(relation) statement  
while(relation) statement  
for(expression1; relation; expression2) statement
```



or

```
if(relation) {statements}  
while(relation) {statements}  
for(expression1; relation; expression2) {statements}
```

A relation in one of the control statements is an expression of the form:

$x > y$

where two expressions are related by one of the following six relational operators:

|    |                          |
|----|--------------------------|
| <  | less than                |
| >  | greater than             |
| <= | less than or equal to    |
| >= | greater than or equal to |
| == | equal to                 |
| != | not equal to             |

**Beware** of using "=" instead of "==" as a relational operator. Unfortunately, both of these are legal, so there will be no diagnostic message, but "=" will not do a comparison.

The **if** statement causes execution of its range if and *only if* the relation is true. Then control passes to the next statement in sequence.

The **while** statement causes execution of its range repeatedly as long as the relation is true. The relation is tested before each execution of its range; and if the relation is false, control passes to the next statement beyond the range of the **while** statement.

The **for** statement begins by executing **expression1**. Then the relation is tested; and, if true, the statements in the range of the **for** are executed. Then **expression2** is executed. The relation is then tested, etc. The typical use of the **for** statement is for a controlled iteration, as in the statement:

```
for(i=1; i<=10; i=i+1) i
```

which will print the integers from one to ten. Following are some examples of the use of the control statements:

```
define f(n){  
  auto i, x  
  x=1  
  for(i=1; i<=n; i=i+1) x=x*i  
  return(x)  
}
```

The input line

$f(a)$



will print "a" factorial if "a" is a positive integer. The following is the definition of a function which will compute values of the binomial coefficient (m and n are assumed to be positive integers):

```
define b(n,m){
  auto x, j
  x=1
  for(j=1; j<=m; j=j+1) x=x*(n-j+1)/j
  return(x)
}
```

The following function computes values of the exponential function by summing the appropriate series without regard for possible truncation errors:

```
scale = 20
define e(x){
  auto a, b, c, d, n
  a = 1
  b = 1
  c = 1
  d = 0
  n = 1
  while(1==1){
    a = a*x
    b = b*n
    c = c + a/b
    n = n + 1
    if(c==d) return(c)
    d = c
  }
}
```

## ADDITIONAL FEATURES

There are some additional language features that every user should know.

Normally, statements are typed one to a line. It is also permissible, however, to type several statements on a line by separating the statements by semicolons.

If an assignment statement is parenthesized, it then has a value; and it can be used anywhere that an expression can. For example, the input line

```
(x=y+17)
```

not only makes the indicated assignment, but also prints the resulting value.

Following is an example of a use of the value of an assignment statement even when it is not parenthesized. The input line

```
x = a[i=i+1]
```

causes a value to be assigned to x and also increments i before it is used as a subscript.



The following constructs work in BC in exactly the same manner as they do in the C language. Refer to Appendix 7.1 or the C language programming documents for more details.

|                     |                |                        |
|---------------------|----------------|------------------------|
| <code>x=y=z</code>  | is the same as | <code>x=(y=z)</code>   |
| <code>x =+ y</code> | "              | <code>x = x+y</code>   |
| <code>x -= y</code> | "              | <code>x = x-y</code>   |
| <code>x *= y</code> | "              | <code>x = x*y</code>   |
| <code>x /= y</code> | "              | <code>x = x/y</code>   |
| <code>x %= y</code> | "              | <code>x = x%y</code>   |
| <code>x ^= y</code> | "              | <code>x = x^y</code>   |
| <code>x++</code>    | "              | <code>(x=x+1)-1</code> |
| <code>x--</code>    | "              | <code>(x=x-1)+1</code> |
| <code>++x</code>    | "              | <code>x = x+1</code>   |
| <code>--x</code>    | "              | <code>x = x-1</code>   |

**Warning:** In some of these constructions, spaces are significant. There is a real difference between `x=-y` and `x= -y`. The first replaces `x` by `x-y` and the second by `-y`.

The following are three important things to remember when using BC programs:

- To exit a BC program, type **quit**.
- There is a comment convention identical to that of the C language. Comments begin with `/*` and end with `*/`.
- There is a library of math functions which may be obtained by typing at command level:

`bc -l`

This command will load a set of library functions which includes sine (`s`), cosine (`c`), arctangent (`a`), natural logarithm (`l`), exponential (`e`), and Bessel functions of integer order [`j(n,x)`]. The library sets the scale to 20, but it can be reset to another value.

If you type

`bc file ...`

the BC program will read and execute the named file or files before accepting commands from the keyboard. In this way, programs and function definitions may be loaded.



## APPENDIX 7.1

## NOTATION

In the following pages, syntactic categories are in *italics* and literals are in **bold**. Material in brackets "[ ]" is optional.

## TOKENS

Tokens consist of keywords, identifiers, constants, operators, and separators. Token separators may be blanks, tabs, or comments. New-line characters or semicolons separate statements.

## A. Comments

Comments are introduced by the characters **/\*** and terminated by **\*/**.

## B. Identifiers

There are three kinds of identifiers; ordinary, array, and function. All three types consist of single lowercase letters. Array identifiers are followed by square brackets, possibly enclosing an expression describing a subscript. Arrays are singly dimensioned and may contain up to 2048 elements. Indexing begins at zero so an array may be indexed from 0 to 2047. Subscripts are truncated to integers. Function identifiers are followed by parentheses, possibly enclosing arguments. The three types of identifiers do not conflict. A program can have a variable named **x**, an array named **x**, and a function named **x**; all of which are separate and distinct.

## C. Keywords

The following are reserved keywords:

|        |        |
|--------|--------|
| ibase  | if     |
| obase  | break  |
| scale  | define |
| sqrt   | auto   |
| length | return |
| while  | quit   |
| for    |        |

## D. Constants

Constants consist of arbitrarily long numbers with an optional decimal point. The hexadecimal digits A through F are also recognized as digits with values 10 through 15, respectively.

## EXPRESSIONS

The value of an expression is printed unless the main operator is an assignment. Precedence is the same as the order of presentation here with highest appearing first. Left or right associativity, where applicable, is discussed with each operator.

## A. Primitive Expressions

## Named Expressions

Named expressions are places where values are stored. Simply stated, named expressions are legal on the left side of an assignment. The value of a named expression is the value stored in the place named.



### *identifiers*

Simple identifiers are named expressions. They have an initial value of zero.

### *array-name[expression]*

Array elements are named expressions. They have an initial value of zero.

### *scale, ibase, and obase*

The internal registers **scale**, **ibase**, and **obase** are all named expressions. The **scale** register is the number of digits after the decimal point to be retained in arithmetic operations. It has an initial value of zero. The **ibase** and **obase** registers are the input and output number radix, respectively. Both **ibase** and **obase** have initial values of ten.

### Function Calls

#### *function-name([expression[,expression..]])*

A function call consists of a function name followed by parentheses containing a comma-separated list of expressions, which are the function arguments. A whole array passed as an argument is specified by the array name followed by empty square brackets. All function arguments are passed by value. As a result, changes made to the formal parameters have no effect on the actual arguments. If the function terminates by executing a return statement, the value of the function is the value of the expression in the parentheses of the **return** statement or is zero if no expression is provided or if there is no **return** statement.

#### *sqrt(expression)*

The result is the square root of the expression. The result is truncated in the least significant decimal place. The scale of the result is the scale of the expression or the value of **scale**, whichever is larger.

#### *length(expression)*

The result is the total number of significant decimal digits in the expression. The scale of the result is zero.

#### *scale(expression)*

The result is the scale of the expression. The scale of the result is zero.

### Constants

Constants are primitive expressions.

### Parentheses

An expression surrounded by parentheses is a primitive expression. The parentheses are used to alter the normal precedence.

### B. Unary Operators

The unary operators bind right to left.

#### *-expression*

The result is the negative of the expression.



***++named-expression***

The named expression is incremented by one. The result is the value of the named expression after incrementing.

***--named-expression***

The named expression is decremented by one. The result is the value of the named expression after decrementing.

***named-expression++***

The named expression is incremented by one. The result is the value of the named expression before incrementing.

***named-expression--***

The named expression is decremented by one. The result is the value of the named expression before decrementing.

**C. Exponentiation Operator**

The exponentiation operator binds right to left.

***expression ^ expression***

The result is the first expression raised to the power of the second expression. The second expression must be an integer. If  $a$  is the scale of the left expression and  $b$  is the absolute value of the right expression, then the scale of the result is

$$\min(axb, \max(\text{scale}, a))$$

**D. Multiplicative Operators**

The operators  $*$ ,  $/$ , and  $\%$  bind left to right.

***expression \* expression***

The result is the product of the two expressions. If  $a$  and  $b$  are the scales of the two expressions, then the scale of the result is

$$\min(a+b, \max(\text{scale}, a, b))$$

***expression / expression***

The result is the quotient of the two expressions. The scale of the result is the value of **scale**.

***expression % expression***

The  $\%$  operator produces the remainder of the division of the two expressions. More precisely,  $a \% b$  is  $a - a/b * b$ .

The scale of the result is the sum of the scale of the divisor and the value of **scale**.



### E. Additive Operators

The additive operators bind left to right.

*expression + expression*

The result is the sum of the two expressions. The scale of the result is the maximum of the scales of the expressions.

*expression - expression*

The result is the difference of the two expressions. The scale of the result is the maximum of the scales of the expressions.

### F. Assignment Operators

The assignment operators bind right to left.

*named-expression = expression*

This expression results in assigning the value of the expression on the right to the named expression on the left.

*named-expression =+ expression*

*named-expression =- expression*

*named-expression =\* expression*

*named-expression =/ expression*

*named-expression =% expression*

*named-expression ^= expression*

The result of the above expressions is equivalent to "named expression = named expression OP expression", where OP is the operator after the = sign.

### RELATIONAL OPERATORS

Unlike all other operators, the relational operators are only valid as the object of an if or while statement or inside a for statement.

*expression < expression*

*expression > expression*

*expression <= expression*

*expression >= expression*

*expression == expression*

*expression != expression*

### STORAGE CLASSES

There are only two storage classes in BC—global and automatic (local). Only identifiers that are to be local to a function need be declared with the **auto** command. The arguments to a function are local to the function. All other identifiers are assumed to be global and available to all functions. All identifiers, global and local, have initial values of zero. Identifiers declared as **auto** are allocated on entry to the function and released on returning from the function. They therefore do not retain values between function calls. The **auto** arrays are specified by the array name followed by empty square brackets.



Automatic variables in BC do not work in exactly the same way as in C language. On entry to a function, the old values of the names that appear as parameters and as automatic variables are pushed onto a stack. Until return is made from the function, reference to these names refers only to the new values.

## STATEMENTS

Statements must be separated by semicolon or new line. Except where altered by control statements, execution is sequential.

### A. Expression Statements

When a statement is an expression unless the main operator is an assignment, the value of the expression is printed, followed by a new-line character.

### B. Compound Statements

Statements may be grouped together and used when one statement is expected by surrounding them with braces { }.

### C. Quoted String Statements

The following statement prints the string inside the quotes.

" any string "

### D. The "if" Statement

*if(relation)statement*

The substatement is executed if the relation is true.

### E. The "while" Statement

*while(relation)statement*

The while statement is executed while the relation is true. The test occurs before each execution of the statement.

### F. The "for" Statement

*for(expression, relation, expression)statement*

The for statement is the same as

```
first-expression
while(relation) {
    statement
    last-expression
}
```

All three expressions must be present.

### G. The "break" Statement

*break*



The **break** statement causes termination of a **for** or **while** statement.

H. The "auto" Statement

```
auto identifier[,identifier]
```

The **auto** statement causes the values of the identifiers to be pushed down. The identifiers can be ordinary identifiers or array identifiers. Array identifiers are specified by following the array name by empty square brackets. The **auto** statement must be the first statement in a function definition.

I. The "define" Statement

```
define([parameter[,parameter...]]){  
    statements}
```

The **define** statement defines a function. The parameters may be ordinary identifiers or array names. Array names must be followed by empty square brackets.

J. The "return" Statement

```
return  
return(expression)
```

The **return** statement causes termination of a function, popping of its auto variables on the stack, and specifies the result of the function. The first form is equivalent to **return(0)**. The result of the function is the result of the expression in parentheses.

K. The "quit" Statement

The **quit** statement stops execution of a BC program and returns control to the UNIX software when it is first encountered. Because it is not treated as an executable statement, it cannot be used in a function definition or in an **if**, **for**, or **while** statement.



## 8. INTERACTIVE DESK CALCULATOR (DC)

### GENERAL

The DC program is an interactive desk calculator program implemented on the UNIX operating system to do arbitrary-precision integer arithmetic. It has provisions for manipulating scaled fixed-point numbers and for input and output in bases other than decimal.

The size of numbers that can be manipulated by DC is limited only by available core storage. On typical implementations of the UNIX system, the size of numbers that can be handled varies from several hundred on the smallest systems to several thousand on the largest.

The DC program works like a stacking calculator using reverse Polish notation. Ordinarily, DC operates on decimal integers; but an input base, output base, and a number of fractional digits to be maintained can be specified.

A language called BC has been developed which accepts programs written in the familiar style of higher-level programming languages and compiles the output which is interpreted by DC. Some of the commands described below were designed for the compiler interface and are not easy for a human user to manipulate.

Numbers that are typed into DC are put on a pushdown stack. The DC commands work by taking the top number or two off the stack, performing the desired operation, and pushing the result on the stack. If an argument is given, input is taken from that file until its end, then from the standard input.

### DC COMMANDS

Any number of commands are permitted on a line. Blanks and new-line characters are ignored except within numbers and in places where a register name is expected.

The following constructions are recognized:

number

The value of the number is pushed onto the main stack. A number is an unbroken string of digits 0 through 9 and uppercase letters A through F (treated as digits with values 10 through 15, respectively). The number may be preceded by an underscore ( `_` ) to input a negative number. Numbers may contain decimal points.

`+ - * / % ^`

The top two values on the stack are added (+), subtracted (-), multiplied (\*), divided (/), remaindered (%), or exponentiated (^). The two entries are popped off the stack, and the result is pushed on the stack in their place. The result of a division is an integer truncated toward zero. An exponent must not have any digits after the decimal point.

`sx`

The top of the main stack is popped and stored into a register named `x`, where `x` may be any character. If `s` is uppercase, `x` is treated as a stack; and the value is pushed onto it. Any character, even blank or new line, is a valid register name.

`lx`

The value in register `x` is pushed onto the stack. The register `x` is not altered. If `l` is uppercase, register `x` is treated as a stack, and its top value is popped onto the main stack. All registers start with empty value which is treated as a zero by the command `l` and is treated as an error by the command `L`.



d

The top value on the stack is duplicated.

p

The top value on the stack is printed. The top value remains unchanged.

f

All values on the stack and in registers are printed.

x

Treats the top element of the stack as a character string, removes it from the stack, and executes it as a string of DC commands.

[ ... ]

Puts the bracketed character string onto the top of the stack.

q

Exits the program. If executing a string, the recursion level is popped by two. If q is uppercase, the top value on the stack is popped; and the string execution level is popped by that value.

<x >x =x !<x !>x !=x

The top two elements of the stack are popped and compared. Register x is executed if they obey the stated relation. Exclamation point is negation.

v

Replaces the top element on the stack by its square root. The square root of an integer is truncated to an integer.

!

Interprets the rest of the line as a UNIX software command. Control returns to DC when the command terminates.

c

All values on the stack are popped; the stack becomes empty.

i

The top value on the stack is popped and used as the number radix for further input. If i is uppercase, the value of the input base is pushed onto the stack. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 or greater than 16.

o

The top value on the stack is popped and used as the number radix for further output. If o is uppercase, the value of the output base is pushed onto the stack.



k

The top of the stack is popped, and that value is used as a scale factor that influences the number of decimal places that are maintained during multiplication, division, and exponentiation. The scale factor must be greater than or equal to zero and less than 100. If k is uppercase, the value of the scale factor is pushed onto the stack.

z

The value of the stack level is pushed onto the stack.

?

A line of input is taken from the input source (usually the console) and executed.

### INTERNAL REPRESENTATION OF NUMBERS

Numbers are stored internally using a dynamic storage allocator. Numbers are kept in the form of a string of digits to the base 100 stored one digit per byte (centennial digits). The string is stored with the low-order digit at the beginning of the string. For example, the representation of 157 is 57,1. After any arithmetic operation on a number, care is taken that all digits are in the range 0 to 99 and that the number has no leading zeros. The number zero is represented by the empty string.

Negative numbers are represented in the 100s complement notation, which is analogous to twos complement notation for binary numbers. The high-order digit of a negative number is always -1 and all other digits are in the range 0 to 99. The digit preceding the high-order -1 digit is never a 99. The representation of -157 is 43,98,-1. This is called the canonical form of a number. The advantage of this kind of representation of negative numbers is ease of addition. When addition is performed digit by digit, the result is formally correct. The result need only be modified, if necessary, to put it into canonical form.

Because the largest valid digit is 99 and the byte can hold numbers twice that large, addition can be carried out and the handling of carries done later when it is convenient.

An additional byte is stored with each number beyond the high-order digit to indicate the number of assumed decimal digits after the decimal point. The representation of .001 is 1,3 where the scale has been italicized to emphasize the fact that it is not the high-order digit. The value of this extra byte is called the **scale factor** of the number.

### THE ALLOCATOR

The DC program uses a dynamic string storage allocator for all of its internal storage. All reading and writing of numbers internally is through the allocator. Associated with each string in the allocator is a 4-word header containing pointers to the beginning of the string, the end of the string, the next place to write, and the next place to read. Communication between the allocator and DC is via pointers to these headers.

The allocator initially has one large string on a list of free strings. All headers except the one pointing to this string are on a list of free headers. Requests for strings are made by size. The size of the string actually supplied is the next higher power of two. When a request for a string is made, the allocator first checks the free list to see if there is a string of the desired size. If none is found, the allocator finds the next larger free string and splits it repeatedly until it has a string of the right size. Leftover strings are put on the free list. If there are no larger strings, the allocator tries to combine smaller free strings into larger ones. Since all strings are the result of splitting large strings, each string has a neighbor that is next to it in core and, if free, can be combined with it to make a string twice as long.

If a string of the proper length can not be found, the allocator asks the system for more space. The amount of space on the system is the only limitation on the size and number of strings in DC. If the allocator runs out of headers at any time in the process of trying to allocate a string, it also asks the system for more space.



There are routines in the allocator for reading, writing, copying, rewinding, forward spacing, and backspacing strings. All string manipulation is done using these routines.

The reading and writing routines increment the read pointer or write pointer so that the characters of a string are read or written in succession by a series of read or write calls. The write pointer is interpreted as the end of the information-containing portion of a string and a call to read beyond that point returns an end of string indication. An attempt to write beyond the end of a string causes the allocator to allocate a larger space and then copy the old string into the larger block.

## INTERNAL ARITHMETIC

All arithmetic operations are done on integers. The operands (or operand) needed for the operation are popped from the main stack and their scale factors stripped off. Zeros are added or digits removed as necessary to get a properly scaled result from the internal arithmetic routine. For example, if the scale of the operands is different and decimal alignment is required, as it is for addition, zeros are appended to the operand with the smaller scale. After performing the required arithmetic operation, the proper scale factor is appended to the end of the number before it is pushed on the stack.

A register called **scale** plays a part in the results of most arithmetic operations. The **scale** register limits the number of decimal places retained in arithmetic computations. The **scale** register may be set to the number on the top of the stack truncated to an integer with the **k** command. The **K** command may be used to push the value of **scale** on the stack. The value of **scale** must be greater than or equal to 0 and less than 100. The descriptions of the individual arithmetic operations will include the exact effect of **scale** on the computations.

## ADDITION AND SUBTRACTION

The scales of the two numbers are compared and trailing zeros are supplied to the number with the lower scale to give both numbers the same scale. The number with the smaller scale is multiplied by 10 if the difference of the scales is odd. The scale of the result is then set to the larger of the scales of the two operands.

Subtraction is performed by negating the number to be subtracted and proceeding as in addition.

The addition is performed digit by digit from the low order end of the number. The carries are propagated in the usual way. The resulting number is brought into canonical form, which may require stripping of leading zeros, or for negative numbers, replacing the high-order configuration 99,-1 by the digit -1. In any case, digits which are not in the range 0 through 99 must be brought into that range, propagating any carries or borrows that result.

## MULTIPLICATION

The scales are removed from the two operands and saved. The operands are both made positive. Then multiplication is performed in a digit by digit manner that exactly follows the hand method of multiplying. The first number is multiplied by each digit of the second number, beginning with its low order digit. The intermediate products are accumulated into a partial sum which becomes the final product. The product is put into the canonical form and its sign is computed from the signs of the original operands.

The scale of the result is set equal to the sum of the scales of the two operands. If that scale is larger than the internal register **scale** and also larger than both of the scales of the two operands, then the scale of the result is set equal to the largest of these three last quantities.

## DIVISION

The scales are removed from the two operands. Zeros are appended, or digits are removed from the dividend to make the scale of the result of the integer division equal to the internal quantity **scale**. The signs are removed and saved.



Division is performed much as it would be done by hand. The difference of the lengths of the two numbers is computed. If the divisor is longer than the dividend, zero is returned. Otherwise, the top digit of the divisor is divided into the top two digits of the dividend. The result is used as the first (high-order) digit of the quotient. If it turns out to be one unit too low, the next trial quotient will be larger than 99; and this will be adjusted at the end of the process. The trial digit is multiplied by the divisor, the result subtracted from the dividend, and the process is repeated to get additional quotient digits until the remaining dividend is smaller than the divisor. At the end, the digits of the quotient are put into the canonical form, with propagation of carry as needed. The sign is set from the sign of the operands.

#### REMAINDER

The division routine is called and division is performed exactly as described. The quantity returned is the remains of the dividend at the end of the divide process. Since division truncates toward zero, remainders have the same sign as the dividend. The scale of the remainder is set to the maximum of the scale of the dividend and the scale of the quotient plus the scale of the divisor.

#### SQUARE ROOT

The scale is removed from the operand. Zeros are added if necessary to make the integer result have a scale that is the larger of the internal quantity scale and the scale of the operand. The method used to compute the square root is Newton's method with successive approximations by the rule

$$X_{n+1} = \frac{1}{2} \left( X_n + \frac{Y}{X_n} \right)$$

The initial guess is found by taking the interger square root of the top two digits.

#### EXPONENTIATION

Only exponents with zero scale factor are handled. If the exponent is zero, then the result is 1. If the exponent is negative, then it is made positive; and the base is divided into one. The scale of the base is removed.

The integer exponent is viewed as a binary number. The base is repeatedly squared, and the result is obtained as a product of those powers of the base that correspond to the positions of the one-bits in the binary representation of the exponent. Enough digits of the result are removed to make the scale of the result the same as if the indicated multiplication had been performed.

#### INPUT CONVERSION AND BASE

Numbers are converted to the internal representation as they are read in. The scale stored with a number is simply the number of fractional digits input. Negative numbers are indicated by preceding the number with an underscore ( \_ ). The hexadecimal digits A through F correspond to the numbers 10 through 15 regardless of input base. The i command can be used to change the base of the input numbers. This command pops the stack, truncates the resulting number to an integer, and uses it as the input base for all further input. The input base (ibase) is initialized to 10 (decimal) but may, for example, be changed to 8 or 16 for octal or hexadecimal to decimal conversions. The command I will push the value of the input base on the stack.

#### OUTPUT COMMANDS

The command p causes the top of the stack to be printed. It does not remove the top of the stack. All of the stack and internal registers can be output by typing the command f. The o command can be used to change the output base (obase). This command uses the top of the stack, truncated to an integer as the base for all further output. The output base is initialized to 10 (decimal). It will work correctly for any base. The command O pushes the value of the output base on the stack.



## OUTPUT FORMAT AND BASE

The input and output bases only affect the interpretation of numbers on input and output; they have no effect on arithmetic computations. Large numbers are output with 70 characters per line; a backslash ( \ ) indicates a continued line. All choices of input and output bases work correctly, although not all are useful. A particularly useful output base is 100000, which has the effect of grouping digits in fives. Bases of 8 and 16 can be used for decimal-octal or decimal-hexadecimal conversions.

## INTERNAL REGISTERS

Numbers or strings may be stored in internal registers or loaded on the stack from registers with the commands **s** and **l**. The command **sx** pops the top of the stack and stores the result in register **x**. The **x** can be any character. The command **lx** puts the contents of register **x** on the top of the stack. The **l** command has no effect on the contents of register **x**. The **s** command, however, is destructive.

## STACK COMMANDS

The command **c** clears the stack. The command **d** pushes a duplicate of the number on the top of the stack onto the stack. The command **z** pushes the stack size on the stack. The command **X** replaces the number on the top of the stack with its scale factor. The command **Z** replaces the top of the stack with its length.

## SUBROUTINE DEFINITIONS AND CALLS

Enclosing a string in brackets "[ ]" pushes the ASCII string on the stack. The **q** command quits or in executing a string pops the recursion levels by two.

## INTERNAL REGISTERS—PROGRAMMING DC

The load and store commands, together with "[ ]" to store strings, the **x** command to execute, and the testing commands (**<**, **>**, **=**, **!<**, **!>**, **!=**), can be used to program **DC**. The **x** command assumes the top of the stack is a string of **DC** commands and executes it. The testing commands compare the top two elements on the stack and if the relation holds execute the register that follows the relation. For example, to print the numbers 0 through 9:

```
[lip1+ si li10>a]sa
0si lax
```

## PUSHDOWN REGISTERS AND ARRAYS

These commands were designed for use by a compiler, not directly by programmers. They involve pushdown registers and arrays. In addition to the stack that commands work on, **DC** can be thought of as having individual stacks for each register. These registers are operated on by the commands **S** and **L**. **Sx** pushes the top value of the main stack onto the stack for the register **x**. **Lx** pops the stack for register **x** and puts the result on the main stack. The commands **s** and **l** also work on registers but not as pushdown stacks. The command **l** does not affect the top of the register stack, but **s** destroys what was there before.

The commands to work on arrays are **:** and **;**. The command **:x** pops the stack and uses this value as an index into the array **x**. The next element on the stack is stored at this index in **x**. An index must be greater than or equal to 0 and less than 2048. The command **;x** loads the main stack from the array **x**. The value on the top of the stack is the index into the array **x** of the value to be loaded.

## MISCELLANEOUS COMMANDS

The command **!** interprets the rest of the line as a UNIX software command and passes it to the UNIX operating system to execute. One other compiler command is **Q**. This command uses the top of the stack as the number of levels of recursion to skip.



## DESIGN CHOICES

The real reason for the use of a dynamic storage allocator was that a general purpose program could be (and in fact has been) used for a variety of other tasks. The allocator has some value for input and for compiling (i.e., the bracket [...] commands) where it cannot be known in advance how long a string will be. The result was that at a modest cost in execution time, all considerations of string allocation and sizes of strings were removed from the remainder of the program and debugging was made easier. The allocation method used wastes approximately 25 percent of available space.

The choice of 100 as a base for internal arithmetic seemingly has no compelling advantage. Yet the base cannot exceed 127 because of hardware limitations and at the cost of 5 percent in space, debugging was made a great deal easier and decimal output was made much faster.

The reason for a stack-type arithmetic design was to permit all DC commands from addition to subroutine execution to be implemented in essentially the same way. The result was a considerable degree of logical separation of the final program into modules with very little communication between modules.

The rationale for the lack of interaction between the scale and the bases was to provide an understandable means of proceeding after a change of base or scale when numbers had already been entered. An earlier implementation which had global notions of scale and base did not work out well. If the value of `scale` were to be interpreted in the current input or output base, then a change of base or scale in the midst of a computation would cause great confusion in the interpretation of the results. The current scheme has the advantage that the value of the input and output bases are only used for input and output, respectively, and they are ignored in all other operations. The value of `scale` is not used for any essential purpose by any part of the program and it is used only to prevent the number of decimal places resulting from the arithmetic operations from growing beyond all bounds.

The design rationale for the choices for the scales of the results of arithmetic were that in no case should any significant digits be thrown away if, on appearances, the user actually wanted them. Thus, if the user wants to add the numbers 1.5 and 3.517, it seemed reasonable to give them the result 5.017 without requiring to unnecessarily specify rather obvious requirements for precision.

On the other hand, multiplication and exponentiation produce results with many more digits than their operands and it seemed reasonable to give as a minimum the number of decimal places in the operands but not to give more than that number of digits unless the user asked for them by specifying a value for `scale`. Square root can be handled in just the same way as multiplication. The operation of division gives arbitrarily many decimal places and there is simply no way to guess how many places the user wants. In this case only, the user must specify a `scale` to get any decimal places at all.

The scale of remainder was chosen to make it possible to recreate the dividend from the quotient and remainder. This is easy to implement; no digits are thrown away.



## NOTES



## 9. LEXICAL ANALYZER GENERATOR (LEX)

### GENERAL

The **Lex** program generator is designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to **Lex**. The **Lex** program generator source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copies the input stream to an output stream, and partitions the input into strings which match the given expressions. As each such string is recognized, the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by **Lex**. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

The user supplies the additional code beyond expression matching needed to complete the tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired.

The **Lex** written code is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called "host languages". Just as general purpose languages can produce code to run on different computer hardware, **Lex** can write code in different host languages. The host language is used for the output code generated by **Lex** and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes **Lex** adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present, the only supported host language is the C language, although FORTRAN (in the form of Ratfor) has been available in the past. The **Lex** generator exists on the UNIX operating system, but the code generated by **Lex** may be taken anywhere the appropriate compilers exist.

The **Lex** program generator turns the user's expressions and actions (called **source** in this section) into the host general purpose language; the generated program is named **yylex**. The **yylex** program will recognize expressions in a stream (called **input** in this section) and perform the specified actions for each expression as it is detected. See Fig. 9.1.

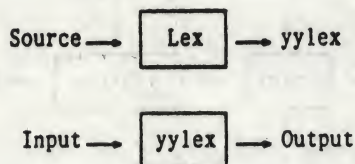


Fig. 9.1 — An Overview of Lex

For an example, consider a program to delete from the input all blanks or tabs at the ends of lines:

```
% %  
[\\t]+$ ;
```



is all that is required. The program contains a `%%` delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression which matches one or more instances of the characters blank or tab (written for visibility, in accordance with the C language convention) which occurs prior to the end of a line. The brackets indicate the character class made of blank and tab; the `+` indicates "one or more ..."; and the `$` indicates "end of line," as in QED. No action is specified, so the program generated by `Lex yylex()` will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

```
%%
[ \t]+$ ;
[ \t]+ printf ( "  " );
```

The coded instructions generated for this source will scan for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a new-line character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule matches all remaining strings of blanks or tabs.

The `Lex` program generator can be used alone for simple transformations or for analysis and statistics gathering on a lexical level. The `Lex` generator can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface `Lex` and `yacc`. The `Lex` program recognizes only regular expressions; `yacc` writes parsers that accept a large class of context free grammars but require a lower level analyzer to recognize input tokens. Thus, a combination of `Lex` and `yacc` is often appropriate. When used as a preprocessor for a later parser generator, `Lex` is used to partition the input stream; and the parser generator assigns structure to the resulting pieces. The flow of control in such a case is shown in Fig. 9.2. Additional programs, written by other generators or by hand, can be added easily to programs written by `Lex`. The `yacc` compiler users will realize that the name `yylex` is what `yacc` expects its lexical analyzer to be named, so that the use of this name by `Lex` simplifies interfacing.

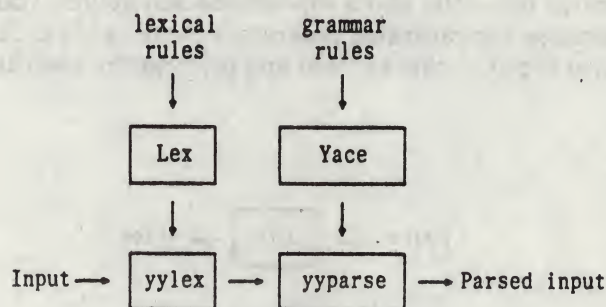


Fig. 9.2 — Lex With Yacc

In the program written by `Lex`, the user's fragments (representing the actions to be performed as each regular expression is found) are gathered as cases of a switch. The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions or to add subroutines outside this action routine.

The `Lex` program generator is not limited to source which can be interpreted on the basis of one character look-ahead. For example, if there are two rules, one looking for "ab" and another for "abcdefg," and the input



stream is "abcdefh," Lex will recognize "ab" and leave the input pointer just before "cd ...". Such backup is more costly than the processing of simpler languages.

## LEX SOURCE

The general format of Lex source is

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first %% is required to mark the beginning of the rules. The absolute minimum Lex program is

```
%%
```

(no definitions, no rules) which translates into a program that copies the input to the output unchanged.

In the outline of Lex programs shown above, the rules represent the user's control decisions; they are a table, in which the left column contains regular expressions and the right column contains actions, program fragments to be executed when the expressions are recognized. Thus an individual rule might appear:

```
integer      printf( " found keyword INT " );
```

to look for the string `integer` in the input stream and print the message "found keyword INT" whenever it appears. In this example the host procedural language is C, and the C language library function `printf` is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C language expression, it can just be given on the right side of the line; if it is compound or takes more than a line, it should be enclosed in braces. As a more useful example, suppose it is desired to change a number of words from British to American spelling. The Lex rules such as:

```
colour      printf( " color " );
mechanise   printf( " mechanize" );
petrol      printf( " gas " );
```

would be a start. These rules are not sufficient since the word "petroleum" would become "gaseum".

## LEX REGULAR EXPRESSIONS

The definitions of regular expressions are very similar to those in QED. A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; the regular expression

```
integer
```

matches the string "integer" wherever it appears, and the expression

```
a57D
```

looks for the string "a57D".



## A. Operators

The operator characters are

" \ [ ] ^ - ? . \* + ! ( ) \$ / { } % < >

and if they are to be used as text characters, an escape should be used. The quotation mark operator " indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus:

xyz " ++ "

matches the string "xyz++" when it appears. Note that a part of a string may be quoted. It is harmless, but unnecessary, to quote an ordinary text character; the expression

" xyz++ "

is equivalent to the one above. Thus, by quoting every nonalphanumeric character being used as a text character, the user can avoid remembering the list above of current operator characters and is safe should further extensions to Lex lengthen the list.

An operator character may also be turned into a text character by preceding it with a backslash (\) as in

xyz\+\+

which is another, less readable, equivalent of the above expressions. Another use of the quoting mechanism is to get a blank into an expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within [ ] (see below) must be quoted. Several normal C language escapes with \ are recognized: \n is new line, \t is tab, and \b is backspace. To enter \ itself, use \\. Since new line is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character except blank, tab, new line, and the list of operator characters above is always a text character.

## B. Character Classes

Classes of characters can be specified using the operator pair [ ]. The construction [abc] matches a single character which may be "a", "b", or "c". Within square brackets, most operator meanings are ignored. Only three characters are special: these are \, -, and ^. The - character indicates ranges. For example:

[a-z0-9<>\_]

indicates the character class containing all the lowercase letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using - between any pair of characters which are not both uppercase letters, both lowercase letters, or both digits is implementation dependent and will get a warning message (e.g., [0-z] in ASCII is many more characters than is in EBCDIC). If it is desired to include the character - in a character class, it should be first or last; thus:

[ - + 0-9]

matches all the digits and the two signs.

In character classes, the ^ operator must appear as the first character after the left bracket to indicate that the resulting string is complemented with respect to the computer character set. Thus:

[^abc]

matches all characters except "a", "b", or "c", including all special or control characters; or

[^a-zA-Z]



is any character which is not a letter. The `\` character provides the usual escapes within character class brackets.

### C. Arbitrary Character

To match almost any character, the operator character (dot)

is the class of all characters except new line. Escaping into octal is possible although nonportable

`[\40-\176]`

matches all printable ASCII characters, from octal 40 (blank) to octal 176 (tilde).

### D. Optional Expressions

The operator `?` indicates an optional element of an expression. Thus:

`ab?c`

matches either "ac" or "abc".

### E. Repeated Expressions

Repetitions of classes are indicated by the operators `*` and `+` for example:

`a*`

is any number of consecutive "a" characters, including zero; while:

`a+`

is one or more instances of "a". For example:

`[a-z]+`

is all strings of lowercase letters, and:

`[A-Za-z][A-Za-z0-9]*`

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

### F. Alternation and Grouping

The operator `|` indicates alternation:

`(ab|cd)`

matches either "ab" or "cd". Note that parentheses are used for grouping; although they are not necessary on the outside level,

`ab|cd`



would have sufficed. Parentheses can be used for more complex expressions:

`(abcd+)?(ef)*`

matches such strings as "abefef", "efefef", "cdef", or "cddd"; but not "abc", "abcd", or "abcdef".

#### G. Context Sensitivity

The **Lex** program will recognize a small amount of surrounding context. The two simplest operators for this are `^` and `$`. If the first character of an expression is `^`, the expression will only be matched at the beginning of a line (after a new-line character or at the beginning of the input stream). This can never conflict with the other meaning of `^` (complementation of character classes) since that only applies within the `[ ]` operators. If the very last character is `$`, the expression will only be matched at the end of a line (when immediately followed by new line). The latter operator is a special case of the `/` operator character which indicates trailing context. The expression

`ab/cd`

matches the string "ab" but only if followed by "cd". Thus:

`ab$`

is the same as

`ab/\n`

Left context is handled in **Lex** by "start conditions" as explained later. If a rule is only to be executed when the **Lex** automaton interpreter is in start condition `x`, the rule should be prefixed by

`<x>`

using the angle bracket operator characters. If we considered "being at the beginning of a line" to be start condition `ONE`, then the `^` operator would be equivalent to

`<ONE>`

Start conditions are explained more fully later.

#### H. Repetitions and Definitions

The operators `{ }` specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example:

`{digit}`

looks for a predefined string named "digit" and inserts it at that point in the expression. The definitions are given in the first part of the **Lex** input, before the rules. In contrast,

`a{1,5}`

looks for one to five occurrences of "a".

Finally, initial `%` is special being the separator for **Lex** source segments.



## LEX ACTIONS

When an expression written as above is matched, **Lex** executes the corresponding action. This part describes some features of **Lex** which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus, the **Lex** user who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When **Lex** is being used with **yacc**, this is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

One of the simplest things that can be done is to ignore the input. Specifying a C language null statement, ; as an action causes this result. A frequent rule is

```
[ \t\n] ;
```

which causes the three spacing characters (blank, tab, and new line) to be ignored.

Another easy way to avoid writing actions is the action character ! which indicates that the action for this rule is the action for the next rule. The previous example could also have been written

```
" "      !
" \t"    !
" \n"    ;
```

with the same result although in different style. The quotes around `\n` and `\t` are not required.

In more complex actions, the user will often want to know the actual text that matched some expression like `"[a-z]+"`. The **Lex** program leaves this text in an external character array. Thus, to print the name found, a rule like

```
[a-z]+      printf( " %s " , yytext);
```

will print the string in `yytext[]`. The C language function `printf` accepts a format argument and data to be printed; in this case, the format is "print string" (% indicating data conversion, and `s` indicating string type), and the data are the characters in `yytext[]`. So this places the matched string on the output. This action is so common that it may be written as **ECHO**:

```
[a-z]+      ECHO;
```

is the same as the above. Since the default action is just to print the characters found, one might ask why give a rule, like this one, which merely specifies the default action. Such rules are often required to avoid matching some other rule which is not desired. For example, if there is a rule which matches `read`, it will normally match the instances of `read` contained in `bread` or `readjust`; to avoid this, a rule of the form `"[a-z]+"` is needed. This is explained further below.

Sometimes it is more convenient to know the end of what has been found; hence **Lex** also provides a count `yytext` of the number of characters matched. To count both the number of words and the number of characters in words in the input, the user might write

```
[a-zA-Z]+      {words++; chars += yytext;
```

which accumulates in `chars` the number of characters in the words recognized. The last character in the string matched can be accessed by:

```
yytext[yytext-1]
```



Occasionally, a **Lex** action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, *yymore()* can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string would overwrite the current entry in *yytext*. Second, *yyless(n)* may be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument "n" indicates the number of characters in *yytext* to be retained. Further characters previously matched are returned to the input. This provides the same sort of look ahead offered by the / operator but in a different form.

Example: Consider a language which defines a string as a set of characters between quotation ( " ) marks and provides that to include a ( " ) in a string it must be preceded by a \. The regular expression which matches that is somewhat confusing, so that it might be preferable to write

```
\ "[^"]*" {
    if (yytext[yylen-1] == '\\')
        yymore();
    else
        ... normal user processing
}
```

which will, when faced with a string such as " abc\" def " first match the five characters " abc\"; then the call to *yymore()* will cause the next part of the string, " def to be tacked on the end. Note that the final quote terminating the string should be picked up in the code labeled "normal processing".

The function *yyless()* might be used to reprocess text in various circumstances. Consider the C language problem of distinguishing the ambiguity of "--a". Suppose it is desired to treat this as "--a" but print a message. A rule might be

```
--[a-zA-Z] {
    printf( " Operator (-- ) ambiguous\n " );
    yyless(yylen-1);
    ... action for == ...
}
```

which prints a message, returns the letter after the operator to the input stream, and treats the operator as "--". Alternatively, it might be desired to treat this as "= -a". To do this, just return the minus sign as well as the letter to the input;

```
--[a-zA-Z] {
    printf( " Operator (-- ) ambiguous\n " );
    yyless(yylen-2);
    ... action for = ...
}
```

will perform the other interpretation. Note that the expressions for the two cases might more easily be written

```
--/[A-Za-z]
```

in the first case, and

```
=/[A-Za-z]
```

in the second; no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of "--3", however, makes

```
--/[^\t\n]
```



a still better rule.

In addition to these routines, **Lex** also permits access to the I/O routines it uses. They are:

1. *input()* returns the next input character
2. *output(c)* writes the character "c" on the output
3. *unput(c)* pushes the character "c" back onto the input stream to be read later by *input()*.

By default these routines are provided as macro definitions, but the user can override them and supply private versions. These routines define the relationship between external files and internal characters and must all be retained or modified consistently. They may be redefined to cause input or output to be transmitted to or from strange places including other programs or internal memory. The character set used must be consistent in all routines, a value of zero returned by *input* must mean end of file, and the relationship between *unput* and *input* must be retained or the **Lex** look ahead will not work. The **Lex** program does not look ahead at all if it does not have to, but every rule ending in +, \*, ?, or \$ or containing / implies look ahead. Look ahead is also necessary to match an expression that is a prefix of another expression. See below for a discussion of the character set used by **Lex**. The standard **Lex** library imposes a 100-character limit on backup.

Another **Lex** library routine that the user will sometimes want to redefine is *yywrap()* which is called whenever **Lex** reaches an end of file. If *yywrap* returns a 1, **Lex** continues with the normal wrap up on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, the user should provide a *yywrap* which arranges for new input and returns 0. This instructs **Lex** to continue processing. The default *yywrap* always returns 1.

This routine is also a convenient place to print tables, summaries, etc., at the end of a program. Note that it is not possible to write a normal rule which recognizes end of file; the only access to this condition is through *yywrap*. In fact, unless a private version of *input()* is supplied, a file containing nulls cannot be handled since a value of 0 returned by *input* is taken to be end of file.

### AMBIGUOUS SOURCE RULES

The **Lex** program can handle ambiguous specifications. When more than one expression can match the current input, **Lex** chooses as follows:

1. The longest match is preferred.
2. Among rules which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

integer  
[a-z]+

keyword action ...;  
identifier action ...;

are to be given in that order. If the input is "integers", it is taken as an identifier, because "[a-z]+" matches eight characters while "integer" matches only seven. If the input is "integer", both rules match seven characters, and the keyword rule is selected because it was given first. Anything shorter (e.g., "int") will not match the expression "integer" and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like .\* dangerous. For example:

.\*



might seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input:

'first' quoted string here, 'second' here

the above expression will match:

'first' quoted string here, 'second'

which is probably not what was wanted. A better rule is of the form:

`[^'\n]*`

which, on the above input, will stop after ('first'). The consequences of errors like this are mitigated by the fact that the dot (.) operator will not match new line. Thus expressions like `.*` stop on the current line. Do not try to defeat this with expressions like `[.\n]+` or equivalents; the Lex generated program will try to read the entire input file causing internal buffer overflows.

Note that Lex is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count occurrences of both "she" and "he" in an input text. Some Lex rules to do this might be:

```
she      s++;
he       h++;
\n       |
        ;
```

where the last two rules ignore everything besides "he" and "she". Remember that dot (.) does not include new line. Since "she" includes "he", Lex will normally not recognize the instances of "he" included in "she" since once it has passed a "she" those characters are gone.

Sometimes the user would like to override this choice. The action *REJECT* means "go do the next alternative". It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose the user really wants to count the included instances of "he":

```
she      {s++; REJECT;}
he       {h++; REJECT;}
\n       |
        ;
```

these rules are one way of changing the previous example to accomplish the task. After counting each expression, it is rejected; whenever appropriate, the other expression will then be counted. In this example, of course, the user could note that "she" includes "he" but not vice versa and omit the *REJECT* action on "he". In other cases, it would not be possible to state which input characters were in both classes.

Consider the two rules

```
a[bc]+   { ... ; REJECT;}
a[cd]+   { ... ; REJECT;}
```

If the input is "ab", only the first rule matches, and on "ad" only the second matches. The input string "accb" matches the first rule for four characters and then the second rule for three characters. In contrast, the input "accd" agrees with the second rule for four characters and then the first rule for three.

In general, *REJECT* is useful whenever the purpose of Lex is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other.



Suppose a digram table of the input is desired; normally the digrams overlap, that is the word "the" is considered to contain both "th" and "he". Assuming a 2-dimensional array named *digram[]* to be incremented, the appropriate source is:

```
%%
[a-z][a-z]    {digram[yytext[0]][yytext[1]]++; REJECT;}
;
\n           ;
```

where the *REJECT* is necessary to pick up a letter pair beginning at every character rather than at every other character.

The action *REJECT* does not rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found and *REJECT* executed the user must not have used *unput* to change the characters forthcoming from the input stream. This is the only restriction on the user's ability to manipulate the not-yet-processed input.

## LEX SOURCE DEFINITIONS

Recalling the format of the **Lex** source:

```
{definitions}
%%
{rules}
%%
{user routines}
```

So far only the rules have been described. The user needs additional options though to define variables for use in the program and for use by **Lex**. Variables can go either in the definitions section or in the rules section.

Remember **Lex** is generating the rules into a program. Any source not intercepted by **Lex** is copied into the generated program. There are three classes of such things.

1. Any line not part of a **Lex** rule or action that begins with a blank or tab is copied into the **Lex** generated program. Such source input prior to the first %% delimiter will be external to any function in the code; if it appears immediately after the first %, it appears in an appropriate place for declarations in the function written by **Lex** which contains the actions. This material must look like program fragments and should precede the first **Lex** rule.

Lines that begin with a blank or tab and that contain a comment are passed through to the generated program. This can be used to include comments in either the **Lex** source or the generated code; the comments should follow the host language convention.

2. Anything included between lines containing only %{ and %} is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1 or copying lines that do not look like programs.
3. Anything after the third %% delimiter, regardless of formats, etc., is copied out after the **Lex** output.

Definitions intended for **Lex** are given before the first %% delimiter. Any line in this section not contained between %{ and %} and beginning in column 1 is assumed to define **Lex** substitution strings. The format of such lines is

|      |             |
|------|-------------|
| name | translation |
|------|-------------|



and it causes the string given as a translation to be associated with the name. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be called out by the `{name}` syntax in a rule. Using `{D}` for the digits and `{E}` for an exponent field, for example, abbreviate rules to recognize numbers

```

D          [0-9]
E          [DEde][+-]?{D}+
% %
{D}+      printf( " integer " );
{D}+ "." {D}*({E})?
{D}* "." {D}+({E})?
{D}+{E}   printf( " real " );

```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field. The first requires at least one digit before the decimal point, and the second requires at least one digit after the decimal point. To correctly handle the problem posed by a Fortran expression such as "35.EQ.I", which does not contain a real number, a context-sensitive rule such as:

```
[0-9]+/ "." EQ printf( " integer " );
```

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands including the selection of a host language, a character set table, a list of start conditions, or adjustments to the default size of arrays within **Lex** itself for larger source programs. These possibilities are discussed later.

## USAGE

There are two steps in compiling a **Lex** source program. First, the **Lex** source must be turned into a generated program in the host general purpose language. Then this program must be compiled and loaded usually with a library of **Lex** subroutines. The generated program is on a file named *lex.yy.c*. The I/O library is defined in terms of the C language standard library.

On the UNIX operating system, the library is accessed by the loader flag `-ll`. So an appropriate set of commands is

```
lex source
cc lex.yy.c -ll
```

The resulting program is placed on the usual file *a.out* for later execution. To use **Lex** with **yacc**, see part "LEX AND YACC". Although the default **Lex** I/O routines use the C language standard library, the **Lex** automata themselves do not do so; if private versions of *input*, *output*, and *unput* are given, the library can be avoided.

## LEX AND YACC

To use **Lex** with **yacc**, note that what **Lex** writes is a program named *yylex()*, the name required by **yacc** for its analyzer. Normally, the default main program on the **Lex** library calls this routine; but if **yacc** is loaded and its main program is used, **yacc** will call *yylex()*. In this case, each **Lex** rule should end with

```
return(token);
```

where the appropriate token value is returned. An easy way to get access to **yacc**'s names for tokens is to compile the **Lex** output file as part of the **yacc** output file by placing the line

```
#include "lex.yy.c"
```



in the last section of **yacc** input. Supposing the grammar to be named "good" and the lexical rules to be named "better", the UNIX software command sequence can just be

```
yacc good
lex better
cc y.tab.c -ly -ll
```

The **yacc** library (**-ly**) should be loaded before the **Lex** library to obtain a main program which invokes the **yacc** parser. The generations of **Lex** and **yacc** programs can be done in either order.

### EXAMPLES

As a problem, consider copying an input file while adding three to every positive number divisible by seven. A suitable **Lex** source program follows:

```
% %
int k;
[0-9]+ {
    k = atoi(yytext);
    if (k%7 == 0)
        printf(" %d ", k+3);
    else
        printf(" %d ", k);
}
```

The rule "[0-9]+" recognizes strings of digits; **atoi()** converts the digits to binary and stores the result in "k". The operator % (remainder) is used to check whether "k" is divisible by seven; if it is, "k" is incremented by three as it is written out. It may be objected that this program will alter such input items as "49.63" or "X7". Furthermore, it increments the absolute value of all negative numbers divisible by seven. To avoid this, add a few more rules after the active one, as here:

```
% %
int k;
-?[0-9]+ {
    k = atoi(yytext);
    printf(" %d ", k%7 == 0 ? k+3 : k);
}
-?[0-9.]+ ECHO;
[A-Za-z][A-Za-z0-9]+ ECHO;
```

Numerical strings containing a dot (.) or preceded by a letter will be picked up by one of the last two rules and not changed. The "if-else" has been replaced by a C language conditional expression to save space; the form "a?b:c" means "if a then b else c".



For an example of statistics gathering, here is a program which histograms the lengths of words, where a word is defined as a string of letters:

```

                                int lengs[100];
%%
[a-z]+                          lengs[yyvaleng]++;
.                                |
\n                               ;
%%
yywrap()
{
    int i;
    printf( " Length No. words\n " );
    for(i=0; i<100; i++)
        if (lengs[i] > 0)
            printf( " %5d%10d\n " ,i,lengs[i]);
    return(1);
}

```

This program accumulates the histogram while producing no output. At the end of the input, it prints the table. The final statement "return(1);" indicates that **Lex** is to perform wrap up. If *yywrap* returns zero (false), it implies that further input is available and the program is to continue reading and processing. To provide a *yywrap* that never returns true causes an infinite loop.

#### LEFT CONTEXT SENSITIVITY

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The `^` operator, for example, is a prior context operator recognizing immediately preceding left context just as `$` recognizes immediately following right context. Adjacent left context could be extended to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful since often the relevant left context appeared some time earlier such as at the beginning of a line.

This part describes three means of dealing with different environments: a simple use of flags (when only a few rules change from one environment to another), the use of "start conditions" on rules, and the possibility of making multiple lexical analyzers all run together. In each case, there are rules that recognize the need to change the environment in which the following input text is analyzed and that set a parameter to reflect the change. This may be a flag explicitly tested by the user's action code; this is the simplest way of dealing with the problem since **Lex** is not involved at all. It may be more convenient, however, to have **Lex** remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It will only be recognized when **Lex** is in that start condition. The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers and switching from one to another as desired.

Consider the following problem: copy the input to the output, changing the word "magic" to "first" on every line which began with the letter "a", changing "magic" to "second" on every line which began with the letter "b", and changing "magic" to "third" on every line which began with the letter "c". All other words and all other lines are left unchanged.



These rules are so simple that the easiest way to do this job is with a flag:

```

    int flag;
%%
^a {flag = 'a'; ECHO;}
^b {flag = 'b'; ECHO;}
^c {flag = 'c'; ECHO;}
\n {flag = 0; ECHO;}
magic {
    switch (flag)
    {
        case 'a': printf( " first " ); break;
        case 'b': printf( " second " ); break;
        case 'c': printf( " third " ); break;
        default: ECHO; break;
    }
}

```

should be adequate.

To handle the same problem with start conditions, each start condition must be introduced to **Lex** in the definitions section with a line reading:

```
%Start  name1 name2 ...
```

where the conditions may be named in any order. The word "Start" may be abbreviated to "s" or "S". The conditions may be referenced at the head of a rule with <> brackets;

```
<name1>expression
```

is a rule which is only recognized when **Lex** is in the start condition **name1**. To enter a start condition, execute the action statement

```
BEGIN name1;
```

which changes the start condition to **name1**. To resume the normal state

```
BEGIN 0;
```

resets the initial condition of the **Lex** automaton interpreter. A rule may be active in several start conditions

```
<name1,name2,name3>
```

is a legal prefix. Any rule not beginning with the <> prefix operator is always active.

The same example as before can be written:

```

%START AA BB CC
%%
^a {ECHO; BEGIN AA;}
^b {ECHO; BEGIN BB;}
^c {ECHO; BEGIN CC;}
\n {ECHO; BEGIN 0;}
<AA>magic printf( " first " );
<BB>magic printf( " second " );
<CC>magic printf( " third " );

```

where the logic is exactly the same as in the previous method of handling the problem, but **Lex** does the work rather than the user's code.



## CHARACTER SET

The programs generated by **Lex** handle character I/O only through the routines *input()*, *output()*, and *unput()*. Thus, the character representation provided in these routines is accepted by **Lex** and used to return values in *yytext()*. For internal use, a character is represented as a small integer which, if the standard library is used, has a value equal to the integer value of the bit pattern representing the character on the host computer. Normally, the letter **a** is represented in the same form as the character constant **'a'**. If this interpretation is changed by providing I/O routines that translate the characters, **Lex** must be given a translation table that is in the definitions section and must be bracketed by lines containing only **%T**; the translation table contains lines of the form:

```
{integer} {character string}
```

which indicate the value associated with each character.

## SUMMARY OF SOURCE FORMAT

The general form of a **Lex** source file is

```
{definitions}  
%%  
{rules}  
%%  
{user subroutines}
```

The definitions section contains a combination of:

1. Definitions in the form "name space translation".
2. Included code in the form "space code".
3. Included code in the form:

```
%{  
code  
%}
```

4. Start conditions given in the form:

```
%S name1 name2 ...
```

5. Character set tables in the form:

```
%T  
number space character-string  
...  
%T
```



## 6. Changes to internal array sizes in the form:

%x nnn

where "nnn" is a decimal integer representing an array size and "a" selects the parameter as follows:

| Letter | Parameter                |
|--------|--------------------------|
| p      | positions                |
| n      | states                   |
| e      | tree nodes               |
| a      | transitions              |
| k      | packed character classes |
| o      | output array size        |

Lines in the rules section have the form "expression action" where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in **Lex** use the following operators:

|        |                                                        |
|--------|--------------------------------------------------------|
| x      | the character "x"                                      |
| " x "  | an "x", even if x is an operator.                      |
| \x     | an "x", even if x is an operator.                      |
| [xy]   | the character x or y.                                  |
| [x-z]  | the characters x, y, or z.                             |
| [^x]   | any character but x.                                   |
| .      | any character but new line.                            |
| ^x     | an x at the beginning of a line.                       |
| <y>x   | an x when Lex is in start condition y.                 |
| x\$    | an x at the end of a line.                             |
| x?     | an optional x.                                         |
| x*     | 0,1,2, ... instances of x.                             |
| x+     | 1,2,3, ... instances of x.                             |
| x y    | an x or a y.                                           |
| (x)    | an x.                                                  |
| x/y    | an x but only if followed by y.                        |
| {xx}   | the translation of xx from<br>the definitions section. |
| x{m,n} | m through n occurrences of x                           |

**CAVEATS AND BUGS**

There are pathological expressions that produce exponential growth of the tables when converted to deterministic machines; fortunately, they are rare.

*REJECT* does not rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found, and *REJECT* executed, the user must not have used *unput* to change the characters forthcoming from the input stream. This is the only restriction on the user's ability to manipulate the not-yet-processed input.



## NOTES



## 10. YET ANOTHER COMPLIER—COMPLIER (yacc)

## GENERAL

The yacc program provides a general tool for imposing structure on the input to a computer program. The yacc user prepares a specification of the input process. This includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. The yacc program then generates a function to control the input process. This function, called a *parser*, calls the user-supplied low-level input routine (the *lexical analyzer*) to pick up the basic items (called *tokens*) from the input stream. These tokens are organized according to the input structure rules, called *grammar rules*. When one of these rules has been recognized, then user code supplied for this rule, an *action*, is invoked. Actions have the ability to return values and make use of the values of other actions.

The yacc program is written in a portable dialect of the C language, and the actions and output subroutine are in the C language as well. Moreover, many of the syntactic conventions of yacc follow the C language.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

```
date : month_name day ',' year ;
```

where "date", "month\_name", "day", and "year" represent structures of interest in the input process; presumably, "month\_name", "day", and "year" are defined elsewhere. The comma is enclosed in single quotes. This implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule and have no significance in controlling the input. With proper definitions, the input

```
July 4, 1776
```

might be matched by the rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizes the lower-level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a "terminal symbol", while the structure recognized by the parser is called a "nonterminal symbol". To avoid confusion, terminal symbols will usually be referred to as "tokens".

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
month_name : 'J' 'a' 'n' ;
```

```
month_name : 'F' 'e' 'b' ;
```

```
...
```

```
month_name : 'D' 'e' 'c' ;
```

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and "month\_name" would be a nonterminal symbol. Such low-level rules tend to waste time and space and may complicate the specification beyond the ability of yacc to deal with it. Usually, the lexical analyzer would recognize the month names and return an indication that a "month name" was seen. In this case, "month name" would be a "token".

Literal characters such as a comma must also be passed through the lexical analyzer and are also considered tokens.



Specification files are very flexible. It is relatively easy to add to the above example the rule

date : month '/' day '/' year ;

allowing

7 / 4 / 1776

as a synonym for

July 4, 1776

on input. In most cases, this new rule could be "slipped in" to a working system with minimal effort and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan. Thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data or the continuation of the input process after skipping over the bad data.

In some cases, **yacc** fails to produce a parser when given a set of specifications. For example, the specifications may be self-contradictory, or they may require a more powerful recognition mechanism than that available to **yacc**. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful or by rewriting some of the grammar rules. While **yacc** cannot handle all possible specifications, its power compares favorably with similar systems. Moreover, the constructions which are difficult for **yacc** to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid **yacc** specifications for their input revealed errors of conception or design early in the program development.

The **yacc** program has been extensively used in numerous practical applications, including **lint**, the Portable C Compiler, and a system for typesetting mathematics.

The remainder of this document describes the following subjects as they relate to **yacc**.

- Basic process of preparing a **yacc** specification
- Parser operation
- Handling ambiguities
- Handling operator precedences in arithmetic expressions
- Error detection and recovery
- The operating environment and special features of the parsers **yacc** produces
- Suggestions to improve the style and efficiency of the specifications
- Advanced topics.

In addition there are four appendices. Appendix 10.1 is a brief example, and Appendix 10.2 is a summary of the **yacc** input syntax. Appendix 10.3 gives an example using some of the more advanced features of **yacc**,



and Appendix 10.4 describes mechanisms and syntax no longer actively supported but provided for historical continuity with older versions of **yacc**.

#### BASIC SPECIFICATIONS

Names refer to either tokens or nonterminal symbols. The **yacc** program requires token names to be declared as such. In addition, it is often desirable to include the lexical analyzer as part of the specification file. It may be useful to include other programs as well. Thus, every specification file consists of three sections: the *declarations*, (*grammar*) *rules*, and *programs*. The sections are separated by double percent (%%) marks. (The percent symbol is generally used in **yacc** specifications as an escape character.)

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

when each section is used.

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also. The smallest legal **yacc** specification is

```
%%
rules
```

since the other two sections may be omitted.

Blanks, tabs, and new lines are ignored except that they may not appear in names or multicharacter reserved symbols. Comments may appear wherever a name is legal. They are enclosed in /\* ... \*/, as in C language.

The rules section is made up of one or more grammar rules. A grammar rule has the form

```
A : BODY ;
```

where "A" represents a nonterminal name, and "BODY" represents a sequence of zero or more names and literals. The colon and the semicolon are **yacc** punctuation.

Names may be of arbitrary length and may be made up of letters, dots, underscores, and noninitial digits. Uppercase and lowercase letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes ( ' '). As in C language, the backslash ( \ ) is an escape character within literals, and all the C language escapes are recognized. Thus:

```
'\n' new-line
'\r' return
'\'' single quote ( ' ' )
'\\' backslash ( \ )
'\t' tab
'\b' backspace
'\f' form feed
'\xxx' "xxx" in octal
```



are understood by yacc. For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left-hand side, the vertical bar (|) can be used to avoid rewriting the left-hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A : B C D ;
A : E F ;
A : G ;
```

can be given to yacc as

```
A : B C D
    | E F
    | G
    ;
```

by using the vertical bar. It is not necessary that all grammar rules with the same left side appear together in the grammar rules section although it makes the input much more readable and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated by

```
empty : ;
```

which is understood by yacc.

Names representing tokens must be declared. This is most simply done by writing

```
%token name1 name2 ...
```

in the declarations section. Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, the *start symbol* has particular importance. The parser is designed to recognize the start symbol. Thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left-hand side of the first grammar rule in the rules section. It is possible and desirable to declare the start symbol explicitly in the declarations section using the `%start` keyword

```
%start symbol
```

to define the start symbol.

The end of the input to the parser is signaled by a special token, called the *endmarker*. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen and accepts the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate. Usually the end marker represents some reasonably obvious I/O status, such as "end of file" or "end of record".

## ACTIONS

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens if desired.



An action is an arbitrary C language statement and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements enclosed in curly braces ({} and {}). For example:

```
A : '(' B ')'
    {
        hello( 1, " abc " );
    }
```

and

```
XXX : YYY ZZZ
    {
        printf( " a message\n " );
        flag = 25;
    }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The dollar sign symbol (\$) is used as a signal to yacc in this context.

To return a value, the action normally sets the pseudo-variable \$\$ to some value. For example, the action

```
{ $ = 1; }
```

does nothing but return the value of one.

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables \$1, \$2, ..., which refer to the values returned by the components of the right side of a rule, reading from left to right. If the rule is

```
A : B C D ;
```

then \$2 has the value returned by C, and \$3 the value returned by D. The rule

```
expr : '(' expr ')' ;
```

provides a more concrete example. The value returned by this rule is usually the value of the "expr" in parentheses. This can be indicated by

```
expr : '(' expr ')'
    {
        $ = $2;
    }
```

By default, the value of a rule is the value of the first element in it (\$1). Thus, grammar rules of the form

```
A : B ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of rules. Sometimes, it is desirable to get control before a rule is fully parsed. The yacc permits an action to be written in the middle of a rule as well as at the



end. This rule is assumed to return a value accessible through the usual \$ mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```

A : B
    {
        $$ = 1;
    }
    C
    {
        x = $2;
        y = $3;
    }
    ;

```

the effect is to set x to 1 and y to the value returned by C.

Actions that do not terminate a rule are actually handled by yacc by manufacturing a new nonterminal symbol name and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. The yacc program actually treats the above example as if it had been written

```

$ACT : /* empty */
    {
        $$ = 1;
    }
    ;
A : B $ACT C
    {
        x = $2;
        y = $3;
    }
    ;

```

where \$ACT is an empty action.

In many applications, output is not done directly by the actions. A data structure, such as a parse tree, is constructed in memory and transformations are applied to it before output is generated. Parse trees are particularly easy to construct given routines to build and maintain the tree structure desired. For example, suppose there is a C function *node* written so that the call

```
node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, and returns the index of the newly created node. Then parse tree can be built by supplying actions such as

```

expr : expr '+' expr
    {
        $$ = node( '+', $1, $3 );
    }
    ;

```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section enclosed in the marks %{ and %}. These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example:

```
%{ int variable = 0; %}
```



could be placed in the declarations section making "variable" accessible to all of the actions. The yacc parser uses only names beginning with **yy**. The user should avoid such names.

In these examples, all the values are integers. A discussion of values of other types will be found in the part "ADVANCED TOPICS".

## LEXICAL ANALYSIS

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called *yyllex*. The function returns an integer, the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable *yylval*.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by yacc or the user. In either case, the **# define** mechanism of C language is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name **DIGIT** has been defined in the declarations section of the yacc specification file. The relevant portion of the lexical analyzer might look like

```
yyllex()
{
    extern int yyval;
    int c;
    ...
    c = getchar();
    ...
    switch( c )
    {
        ...
        case '0':
        case '1':
        ...
        case '9':
            yyval = c - '0';
            return( DIGIT );
        ...
    }
    ...
}
```

to return the appropriate token.

The intent is to return a token number of **DIGIT** and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier **DIGIT** will be defined as the token number associated with the token **DIGIT**.

This mechanism leads to clear, easily modified lexical analyzers. The only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C language or the parser. For example, the use of token names **if** or **while** will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name **error** is reserved for error handling and should not be used naively.

As mentioned above, the token numbers may be chosen by yacc or the user. In the default situation, the numbers are chosen by yacc. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.



To assign a token number to a token (including literals), the first appearance of the token name or literal in the declarations section can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user. Thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the `lex` program. These lexical analyzers are designed to work in close harmony with `yacc` parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. `Lex` can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework and whose lexical analyzers must be crafted by hand.

## PARSER OPERATION

The `yacc` program turns the specification file into a C language program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex and will not be discussed here. The parser itself, however, is relatively simple and understanding how it works will make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by `yacc` consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the *look-ahead* token). The *current state* is always the one on the top of the stack. The states of the finite state machine are given small integer labels. Initially, the machine is in state 0, the stack contains only state 0, and no look-ahead token has been read.

The machine has only four actions available—*shift*, *reduce*, *accept*, and *error*. A step of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a look-ahead token to decide the action to be taken. If it needs one and does not have one, it calls `yylex` to obtain the next token.
2. Using the current state and the look-ahead token if needed, the parser decides on its next action and carries it out. This may result in states being pushed onto the stack or popped off of the stack and in the look-ahead token being processed or left alone.

The *shift* action is the most common action the parser takes. Whenever a shift action is taken, there is always a look-ahead token. For example, in state 56 there may be an action

IF shift 34

which says, in state 56, if the look-ahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The look-ahead token is cleared.

The *reduce* action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right-hand side of a grammar rule and is prepared to announce that it has seen an instance of the rule replacing the right-hand side by the left-hand side. It may be necessary to consult the look-ahead token to decide whether to reduce but usually it is not. In fact, the default action (represented by a dot) is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, and this leads to some confusion. The action

. reduce 18



IF shift 34

Suppose the rule

is being reduced. The reduce action depends on the left-hand symbol (A in this case) and the number of symbols on the right-hand side (three in this case). To reduce, first pop off the top three states from the stack. (In general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing  $x$ ,  $y$ , and  $z$  and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left-hand symbol and an ordinary shift of a token, however, so this action is called a *goto* action. In particular, the look-ahead token is cleared by a shift but is not affected by a *goto*. In any case, the uncovered state contains an entry such as

causing state 20 to be pushed onto the stack and become the current state.

In effect, the reduce action “turns back the clock” in the parse popping the states off the stack to go back to the state where the right-hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right-hand side of the rule is empty, no states are popped off of the stacks. The uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack running in parallel with it holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable "yyval" is copied onto the value stack. After the return from the user code, the reduction is carried out. When the *goto* action is done, the external variable "yyval" is copied onto the value stack. The pseudo-variables \$1, \$2, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The *accept* action indicates that the entire input has been seen and that it matches the specification. This action appears only when the look-ahead token is the endmarker and indicates that the parser has successfully done its job. The *error* action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the look-ahead token, cannot be followed by anything that would result in a legal input. The parser reports an error and attempts to recover the situation and resume parsing. The error recovery (as opposed to the detection of error) will be discussed later.

|         |   |       |       |      |
|---------|---|-------|-------|------|
| % token |   | DING  | DONG  | DELL |
| % %     |   |       |       |      |
| rhyme   | : | sound | place |      |
|         | : |       |       |      |
| sound   | : | DING  | DONG  |      |
|         | : |       |       |      |
| place   | : | DELL  |       |      |
|         | : |       |       |      |

Page 139



When `yacc` is invoked with the `-v` option, a file called `y.output` is produced with a human-readable description of the parser. The `y.output` file corresponding to the above grammar (with some statistics stripped off the end) is

```
state 0
    $accept : _rhyme $end

    DING shift 3
    . error

    rhyme goto 1
    sound goto 2

state 1
    $accept : rhyme_$end

    $end accept
    . error

state 2
    rhyme : sound_place

    DELL shift 5
    . error
    place goto 4

state 3
    sound : DING_DONG

    DONG shift 6
    . error

state 4
    rhyme : sound place_ (1)

    . reduce 1

state 5
    place : DELL_ (3)

    . reduce 3

state 6
    sound : DING DONG_ (2)

    . reduce 2
```

where the actions for each state are specified and there is a description of the parsing rules being processed in each state. The `_` character is used to indicate what has been seen and what is yet to come in each rule. The following input

DING DONG DELL

can be used to track the operations of the parser. Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, *DING*, is read and becomes the look-ahead token. The action in state 0 on *DING* is *shift 3*, state 3 is pushed onto the stack, and



the look-ahead token is cleared. State 3 becomes the current state. The next token, *DONG*, is read and becomes the look-ahead token. The action in state 3 on the token *DONG* is *shift 6*, state 6 is pushed onto the stack, and the look-ahead is cleared. The stack now contains 0, 3, and 6. In state 6 without even consulting the look-ahead, the parser reduces by

sound : DING DONG

which is rule 2. Two states, 6 and 3, are popped off of the stack uncovering state 0. Consulting the description of state 0 (looking for a goto on *sound*),

sound goto 2

is obtained. State 2 is pushed onto the stack and becomes the current state.

In state 2, the next token, *DELL*, must be read. The action is *shift 5*, so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the look-ahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right-hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on *place* (the left side of rule 3) is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on *rhyme* causing the parser to enter state 1. In state 1, the input is read and the endmarker is obtained, indicated by \$end in the *y.output* file. The action in state 1 when the end marker is seen is to accept, successfully ending the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as *DING DONG DONG*, *DING DONG*, *DING DONG DELL DELL*, etc. A few minutes spend with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

### AMBIGUITY AND CONFLICTS

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways. For example, the grammar rule

expr : expr '-' expr

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

expr - expr - expr

the rule allows this input to be structured as either

( expr - expr ) - expr

or as

expr - ( expr - expr )

(The first is called "left association", the second "right association".)

The yacc program detects such ambiguities when it is attempting to build the parser. Given the input

expr - expr - expr



consider the problem that confronts the parser. When the parser has read the second `expr`, the input seen:

`expr - expr`

matches the right side of the grammar rule above. The parser could reduce the input by applying this rule. After applying the rule, the input is reduced to "`expr`" (the left side of the rule). The parser would then read the final part of the input:

`- expr`

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

`expr - expr`

it could defer the immediate application of the rule and continue reading the input until

`expr - expr - expr`

has been seen. It could then apply the rule to the rightmost three symbols reducing them to "`expr`" which results in

`expr - expr`

being left. Now the rule can be reduced once more. The effect is to take the right associative interpretation. Thus, having read

`expr - expr`

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a "shift/reduce conflict". It may also happen that the parser has a choice of two legal reductions. This is called a "reduce/reduce conflict". Note that there are never any shift/shift conflicts.

When there are shift/reduce or reduce/reduce conflicts, `yacc` still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing the choice to make in a given situation is called a "disambiguating rule".

The `yacc` program invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the *earlier* grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred in favor of shifts when there is a choice. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided when possible.

Conflicts may arise because of mistakes in input or logic or because the grammar rules, while consistent, require a more complex parser than `yacc` can construct. The use of actions within rules can also cause conflicts if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate and leads to an incorrect parser. For this reason, `yacc` always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.



In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural and produces slower parsers. Thus, yacc will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider:

```
stat : IF '(' cond ')' stat
      | IF '(' cond ')' stat ELSE stat
      ;
```

which is a fragment from a programming language involving an "if-then-else" statement. In these rules, "IF" and "ELSE" are tokens, "cond" is a nonterminal symbol describing conditional (logical) expressions, and "stat" is a nonterminal symbol describing statements. The first rule will be called the "simple-if" rule and the second the "if-else" rule.

These two rules form an ambiguous construction since input of the form

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be structured according to these rules in two ways

```
IF ( C1 )
{
  IF ( C2 )
    S1
}
ELSE
  S2
```

or

```
IF ( C1 )
{
  IF ( C2 )
    S1
  ELSE
    S2
}
```

where the second interpretation is the one given in most programming languages having this construct. Each "ELSE" is associated with the last preceding "un-ELSE'd" IF. In this example, consider the situation where the parser has seen

```
IF ( C1 ) IF ( C2 ) S1
```

and is looking at the "ELSE". It can immediately reduce by the simple-if rule to get

```
IF ( C1 ) stat
```

and then read the remaining input

```
ELSE S2
```



and reduce

```
IF ( C1 ) stat ELSE S2
```

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, the "ELSE" may be shifted, "S2" read, and then the right-hand portion of

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be reduced by the if-else rule to get

```
IF ( C1 ) stat
```

which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input which is usually desired.

Once again the parser can do two valid things—there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This shift/reduce conflict arises only when there is a particular current input symbol, "ELSE", and particular inputs, such as

```
IF ( C1 ) IF ( C2 ) S1
```

have already been seen. In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of yacc are best understood by examining the verbose (-v) option output file. For example, the output corresponding to the above conflict state might be

```
23: shift/reduce conflict (shift 45, reduce 18) on ELSE
```

```
state 23
```

```
stat : IF ( cond ) stat_ (18)
```

```
stat : IF ( cond ) stat_ELSE stat
```

```
ELSE shift 45
```

```
. reduce 18
```

where the first line describes the conflict—giving the state and the input symbol. The ordinary state description gives the grammar rules active in the state and the parser actions. Recall that the underline marks the portion of the grammar rules which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

```
IF ( cond ) stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is "ELSE", it is possible to shift into state 45. State 45 will have, as part of its description, the line

```
stat : IF ( cond ) stat ELSE_stat
```



since the "ELSE" will have been shifted in this state. Back in state 23 the alternative action, described a dot (.), is to be done if the input symbol is not mentioned explicitly in the above actions. In this case if the input symbol is not "ELSE", the parser reduces to

```
stat : IF '(' cond ')' stat
```

by grammar rule 18.

Once again, notice that the numbers following "shift" commands refer to other states, while the numbers following "reduce" commands refer to grammar rule numbers. In the *y.output* file, the rule numbers are printed after those rules which can be reduced. In most one states, there will be at most reduce action possible in the state, and this will be the default command. The user who encounters unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate.

### PRECEDENCE

There is one common situation where the rules given above for resolving conflicts are not sufficient. This is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of precedence levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
expr : expr OP expr
```

and

```
expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar with many parsing conflicts. As disambiguating rules, the user specifies the precedence or binding strength of all the operators and the associativity of the binary operators. This information is sufficient to allow yacc to resolve the parsing conflicts in accordance with these rules and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a yacc keyword: `%left`, `%right`, or `%nonassoc`, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus:

```
%left '+' '-'
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative and have lower precedence than star and slash, which are also left associative. The keyword `%right` is used to describe right associative operators, and the keyword `%nonassoc` is used to describe operators, like the operator `.LT.` in FORTRAN, that may not associate with themselves. Thus:

```
A .LT. B .LT. C
```

is illegal in FORTRAN and such an operator would be described with the keyword `%nonassoc` in yacc. As an example of the behavior of these declarations, the description

```
%right '='
%left '+' '-'
%left '*' '/'
%%
```



```

expr : expr '=' expr
      | expr '+' expr
      | expr '-' expr
      | expr '*' expr
      | expr '/' expr
      | NAME
      ;

```

might be used to structure the input

$a = b = c*d - e - f*g$

as follows

$a = ( b = ( (c*d) - e ) - (f*g) ) )$

in order to perform the correct precedence of operators. When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation but different precedences. An example is unary and binary “-”. Unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, **%prec**, changes the precedence level associated with a particular grammar rule. The keyword **%prec** appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, the rules

```

%left '+' '-'
%left '*' '/'
%%
expr : expr '+' expr
      | expr '-' expr
      | expr '*' expr
      | expr '/' expr
      | '-' expr %prec '*'
      | NAME
      ;

```

might be used to give unary minus the same precedence as multiplication.

A token declared by **%left**, **%right**, and **%nonassoc** need not be, but may be, declared by **%token** as well.

The precedences and associativities are used by **yacc** to resolve parsing conflicts. They give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule. It is the precedence and associativity of the last token or literal in the body of the rule. If the **%prec** construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.
3. When there is a reduce/reduce conflict or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.



4. If there is a shift/reduce conflict and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by *yacc*. This means that mistakes in the specification of precedences may disguise errors in the input grammar. It is a good idea to be sparing with precedences and use them in an essentially "cookbook" fashion until some experience has been gained. The *y.output* file is very useful in deciding whether the parser is actually doing what was intended.

### ERROR HANDLING

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found. It is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser "restarted" after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, *yacc* provides a simple, but reasonably general, feature. The token name "error" is reserved for error handling. This name can be used in grammar rules. In effect, it suggests places where errors are expected and recovery might take place. The parser pops its stack until it enters a state where the token "error" is legal. It then behaves as if the token "error" were the current look-ahead token and performs the action encountered. The look-ahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
stat : error
```

would mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these. If the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general but difficult to control. Rules such as

```
stat : error ';' ;
```

are somewhat easier. Here, when there is an error, the parser attempts to skip over the statement but will do so by skipping to the next semicolon. All tokens after the error and before the next semicolon cannot be shifted and are discarded. When the semicolon is seen, this rule will be reduced and any "cleanup" action associated with it performed.



Another form of error rule arises in interactive applications where it may be desirable to permit a line to be reentered after an error. The following example

```
input : error '\n'
      {
        printf( " Reenter last line: " );
      }
      input
      {
        $$ = $4;
      }
      ;
```

is one way to do this. There is one potential difficulty with this approach. The parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens and gives no message. This is clearly unacceptable. For this reason, there is a mechanism that can force the parser to believe that an error has been fully recovered from. The statement

```
y yerrok ;
```

in an action resets the parser to its normal mode. The last example can be rewritten as

```
input : error '\n'
      {
        yerrok;
        printf( " Reenter last line: " );
      }
      input
      {
        $$ = $4;
      }
      ;
```

which is somewhat better.

As previously mentioned, the token seen immediately after the "error" symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous look-ahead token must be cleared. The statement

```
y yclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the user, that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by *yylex* would presumably be the first token in a legal statement. The old, illegal token must be discarded, and the error state reset. A rule similar to

```
stat : error
     {
       resynch() ;
       yerrok ;
       yclearin ;
     }
     ;
```



could perform this.

These mechanisms are admittedly crude but do allow for a simple, fairly effective recovery of the parser from many errors. Moreover, the user can get control to deal with the error actions required by other portions of the program.

### THE "yacc" ENVIRONMENT

When the user inputs a specification to *yacc*, the output is a file of C language programs, called *y.tab.c* on most systems. (Due to local file system conventions, the names may differ from installation to installation.) The function produced by *yacc* is called *yyparse()*; it is an integer valued function. When it is called, it in turn repeatedly calls *yylex()*, the lexical analyzer supplied by the user (see part "LEXICAL ANALYSIS") to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) *yyparse()* returns the value 1, or the lexical analyzer returns the end-marker token and the parser accepts. In this case, *yyparse()* returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C language program, a program called *main()* must be defined that eventually calls *yyparse()*. In addition, a routine called *yyerror()* prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using *yacc*, a library has been provided with default versions of *main()* and *yyerror()*. The name of this library is system dependent; on many systems, the library is accessed by a *-ly* argument to the loader. The source code

```
main()
{
    return ( yyparse() );
}
```

and

```
#include <stdio.h>

yyerror(s)
    char *s;
{
    fprintf( stderr, " %s\n" , s );
}
```

show the triviality of these default programs. The argument to *yyerror()* is a string containing an error message, usually the string "syntax error". The average application will want to do better than this. Ordinarily, the program should keep track of the input line number and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the look-ahead token number at the time the error was detected. This may be of some interest in giving better diagnostics. Since the *main()* program is probably supplied by the user (to read arguments, etc.), the *yacc* library is useful only in small projects or in the earliest stages of larger ones.

The external integer variable *yydebug* is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions including a discussion of the input symbols read and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.



## HINTS FOR PREPARING SPECIFICATIONS

This part contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

### A. Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following are a few style hints.

1. Use all uppercase letters for token names and all lowercase letters for nonterminal names. This rule comes under the heading of "knowing who to blame when things go wrong".
2. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.
3. Put all rules with the same left-hand side together. Put the left-hand side in only once and let all following rules begin with a vertical bar.
4. Put a semicolon only after the last rule with a given left-hand side and put the semicolon on a separate line. This allows new rules to be easily added.
5. Indent rule bodies by two tab stops and action bodies by three tab stops.

The example in Appendix 10.1 is written following this style, as are the examples in this section (where space permits). The user must make up his own mind about these stylistic questions. The central problem, however, is to make the rules visible through the morass of action code.

### B. Left Recursion

The algorithm used by the yacc parser encourages so called "left recursive" grammar rules. Rules of the form

```
name : name rest_of_rule ;
```

match this algorithm. These rules such as

```
list  : item
      | list ' ' item
      ;
```

and

```
seq   : item
      | seq item
      ;
```

frequently arise when writing specifications of sequences and lists. In each of these cases, the first rule will be reduced for the first item only; and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as:

```
seq   : item
      | item seq
      ;
```



the parser would be a bit bigger; and the items would be seen and reduced from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification as

```
seq : /* empty */
    | seq item
    ;
```

using an empty rule. Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if yacc is asked to decide which empty sequence it has seen, when it hasn't seen enough to know!

### C. Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally but not within quoted strings, or names might be entered into a symbol table in declarations but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer and set by actions. For example,

```
%{
    int dflag;
}%
... other declarations ...

%%

prog : decls stats
    ;

decls : /* empty */
    {
        dflag = 1;
    }
    | decls declaration
    ;

stats : /* empty */
    {
        dflag = 0;
    }
    | stats statement
    ;

... other rules ...
```

specifies a program which consists of zero or more declarations followed by zero or more statements. The flag "dflag" is now 0 when reading statements and 1 when reading declarations, *except for the first token in the*



*first statement.* This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of "back-door" approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

#### D. Reserved Words

Some programming languages permit the user to use words like "if", which are normally reserved as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of **yacc**. It is difficult to pass information to the lexical analyzer telling it "this instance of *if* is a keyword and that instance is a variable". The user can make a stab at it using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be *reserved*, i.e., be forbidden for use as variable names. There are powerful stylistic reasons for preferring this.

### ADVANCED TOPICS

This part discusses a number of advanced features of **yacc**.

#### A. Simulating Error and Accept in Actions

The parsing actions of error and accept can be simulated in an action by use of macros **YYACCEPT** and **YYERROR**. The **YYACCEPT** macro causes *yyparse()* to return the value 0; **YYERROR** causes the parser to behave as if the current input symbol had been a syntax error; *yerror()* is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple end-markers or context sensitive syntax checking.

#### B. Accessing Values in Enclosing Rules

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit:

```

sent : adj noun verb adj noun
    {
        look at the sentence ...
    }
;

adj : THE
    {
        $$ = THE;
    }
  | YOUNG
    {
        $$ = YOUNG;
    }
;
...
noun : DOG
    {
        $$ = DOG;
    }
;
```



```

| CRONE
| {
|   if( $0 == YOUNG )
|   {
|     printf( "what?/n" );
|   }
|   $$= CRONE;
| }
| ;
...

```

but in this case the digit may be 0 or negative. In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol "noun" in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism will save a great deal of trouble especially when a few combinations are to be excluded from an otherwise regular structure.

### C. Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. The yacc program can also support values of other types including structures. In addition, yacc keeps track of the types and inserts appropriate union member names so that the resulting parser will be strictly type checked. The yacc value stack is declared to be a *union* of the various types of values desired. The user declares the union and associates union member names to each token and nonterminal symbol having a value. When the value is referenced through a \$\$ or \$n construction, yacc will automatically insert the appropriate union name, so that no unwanted conversions will take place. In addition, type checking commands such as lint will be far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union. This must be done by the user since other programs, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where yacc can not easily determine the type.

To declare the union, the user includes

```

%union
{
    body of union ...
}

```

in the declaration section. This declares the yacc value stack and the external variables *yylval* and *yyval* to have type equal to this union. If yacc was invoked with the -d option, the union declaration is copied onto the *y.tab.h* file. Alternatively, the union may be declared in a header file, and a typedef used to define the variable YYSTYPE to represent this union. Thus, the header file might have said

```

typedef union
{
    body of union ...
}
YYSTYPE;

```

instead. The header file must be included in the declarations section by use of `%{ and %}`.



Once YYSTYPE is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

< name >

is used to indicate a union member name. If this follows one of the keywords **%token**, **%left**, **%right**, and **%nonassoc**, the union member name is associated with the tokens listed. Thus, saying

**%left** <optype> '+' '-'

will cause any reference to values returned by these two tokens to be tagged with the union member name *optype*. Another keyword, **%type**, is used to associate union member names with nonterminals. Thus, one might say

**%type** <nodetype> expr stat

to associate the union member *nodetype* with the nonterminal symbols "expr" and "stat".

There remains a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no *a priori* type. Similarly, reference to left context values (such as \$0) leaves **yacc** with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name between < and > immediately after the first \$. The example

```
rule :    aaa
      {
        $<intval>$ = 3;
      }
      bbb
    {
      fun( $<intval>2, $<other>0 );
    }
    ;
```

shows this usage. This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in Appendix 10.3. The facilities in this subsection are not triggered until they are used. In particular, the use of **%type** will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of \$n or \$\$ to refer to something with no defined type is diagnosed. If these facilities are not triggered, the **yacc** value stack is used to hold *int's*, as was true historically.



## APPENDIX 10.1

## A SIMPLE EXAMPLE

This example gives the complete `yacc` specification for a small desk calculator; the desk calculator has 26 registers, labeled "a" through "z", and accepts arithmetic expressions made up of the operators `+`, `-`, `*`, `/`, `%` (mod operator), `&` (bitwise and), `!` (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise, it is. As in C language, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a `yacc` specification, the desk calculator does a reasonable job of showing how precedences and ambiguities are used and demonstrating simple error recovery. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately line by line. Note the way that decimal and octal integers are read in by the grammar rules; this job is probably better done by the lexical analyzer.

```
%{
# include <stdio.h>
# include <ctype.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS /* supplies precedence for unary minus */

%% /* beginning of rules section */

list      :      /* empty */
           |      list stat '\n'
           |      list error '\n'
           {
               yyerrok;
           }
           ;

stat      :      expr
```



```

    {
        printf( "%dn", $1 );
    }
    LETTER '=' expr
    {
        regs[$1] = $3;
    }
;

expr
:   '(' expr ')'
    {
        $$ = $2;
    }
|   expr '+' expr
    {
        $$ = $1 + $3;
    }
|   expr '-' expr
    {
        $$ = $1 - $3;
    }
|   expr '*' expr
    {
        $$ = $1 * $3;
    }
|   expr '/' expr
    {
        $$ = $1 / $3;
    }
|   expr '%' expr
    {
        $$ = $1 % $3;
    }
|   expr '&' expr
    {
        $$ = $1 & $3;
    }
|   expr '|' expr
    {
        $$ = $1 | $3;
    }
|   '-' expr    %prec UMINUS
    {
        $$ = - $2;
    }
|   LETTER
    {
        $$ = regs[$1];
    }
|   number
;

number
:   DIGIT
    {

```



```

        $$ = $1; base = ($1==0) ? 8 : 10;
    }
    |
    number DIGIT
    {
        $$ = bas * $1 + $2;
    }
%% /* start of programs */

yylex( ) /* lexical analysis routine */
{
    /* returns LETTER for a lowercase letter, yylval = 0 through 25*/
    /* returns DIGIT for a digit, yylval = 0 through 9 */
    /* all other characters are returned immediately */

    int c;
    while( (c=getchar( ) ) == ' ' ) /* skip blanks */
        ;

    /* c is now nonblank */

    if( islower( c ) )
    {
        yylval = c - 'a';
        return( LETTER );
    }
    if( isdigit( c ) )
    {
        yylval = c - '0';
        return( DIGIT );
    }
    return( c );
}

```



## APPENDIX 10.2

## YACC INPUT SYNTAX

This appendix has a description of the yacc input syntax as a yacc specification. Context dependencies, etc., are not considered. Ironically, the yacc input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise, it is a continuation of the current rule which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier and decides whether the next token (skipping blanks, new lines, comments, etc.) is a colon. If so, it returns the token C\_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERS but never as part of C\_IDENTIFIERS.

```

/* grammar for the input to Yacc */

/* basic entities */
%token IDENTIFIER /* includes identifiers and literals */
%token C_IDENTIFIER /* identifier (but not literal) followed by colon */
%token NUMBER /* [0-9]+ */

/* reserved words: %type => TYPE, %left => LEFT, etc. */
%token LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token MARK /* the %% mark */
%token LCURL /* the % { mark */
%token RCURL /* the % } mark */

/* ASCII character literals stand for themselves */

%start spec

%%

spec : defs MARK rules tail
      ;

tail : MARK
      {
          In this action, eat up the rest of the file
      }
      /* empty: the second MARK is optional */
      ;

defs : /* empty */
      ;
      defs def
      ;

def : START IDENTIFIER
      | UNION
      {
          Copy union definition to output
      }

```



```

    }
    | LCURL
    {
        Copy C code to output file
        RCURL
    }
    | ndefs rword tag nlist
    ;

rword    : TOKEN
          | LEFT
          | RIGHT
          | NONASSOC
          | TYPE
          ;

tag       : /* empty: union tag is optional */
          | '<' IDENTIFIER '>'
          ;

nlist     : nmno
          | nlist nmno
          | nlist ';' nmno
          ;

nmno      : IDENTIFIER          /* Note: literal illegal with % type */
          | IDENTIFIER NUMBER  /* Note: illegal with % type */
          ;

/* rules section */

rules     : C_IDENTIFIER rbody prec
          | rules rule
          ;

rule      : C_IDENTIFIER rbody prec
          | '!' rbody prec
          ;

rbody     : /* empty */
          | rbody IDENTIFIER
          | rbody act
          ;

act       : '{'
          |
          {
              Copy action, translate $$, etc.
          }
          ;

prec      : /* empty */
          | PREC IDENTIFIER
          | PREC IDENTIFIER act
          | prec ';'
          ;

```



## APPENDIX 10.3

## AN ADVANCED EXAMPLE

This appendix gives an example of a grammar using some of the advanced features. The desk calculator example in Appendix 10.1 is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants; the arithmetic operations  $+$ ,  $-$ ,  $*$ ,  $/$ , unary  $-$ , and  $=$  (assignment); and has 26 floating point variables, "a" through "z". Moreover, it also understands *intervals*, written

$$(x, y)$$

where  $x$  is less than or equal to  $y$ . There are 26 interval valued variables "A" through "Z" that may also be used. The usage is similar to that in Appendix 10.1; assignments return no value and print nothing while expressions print the (floating or interval) value.

This example explores a number of interesting features of yacc and C language. Intervals are represented by a structure consisting of the left and right endpoint values stored as *double's*. This structure is given a type name, INTERVAL, by using *typedef*. The yacc value stack can also contain floating point scalars and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C language. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of YYERROR to handle error conditions—division by an interval containing 0 and an interval presented in the wrong order. The error recovery mechanism of yacc is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (e.g., scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through yacc—18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the two input lines

$$2.5 + (3.5 - 4.)$$

and

$$2.5 + (3.5, 4.)$$

Notice that the 2.5 is to be used in an interval valued expression in the second example, but this fact is not known until the comma is read. By this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator—one when the left operand is a scalar and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflicts will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive but not very general. If there were many kinds of expression types instead of just two, the number of rules needed would increase dramatically and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C language library routine *atof()* is used to do the actual conversion from a character string to a



double precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar provoking a syntax error in the parser and thence error recovery.

```
%{

#include <stdio.h>
#include <ctype.h>

typedef struct interval
{
    double lo, hi;
} INTERVAL;

INTERVAL vmul( ), vdiv( );

double atof( );

double dreg[ 26 ];
INTERVAL vreg[ 26 ];

}%

%start lines

%union
{
    int ival;
    double dval;
    INTERVAL vval;
}

%token <ival> DREG VREG      /* indices into dreg, vreg arrays */
%token <dval> CONST          /* floating point constant */
%type <dval> dexp            /* expression */
%type <vval> vexp            /* interval expression */

/* precedence information about the operators */

%left '+' '-'
%left '*' '/'
%left UMINUS /* precedence for unary minus */

%%

lines : /* empty */
      | lines line
      ;
```



```

line      : dexp '\n'
           {
               printf( "%15.8f\n", $1 );
           }
           {
               vexp '\n'
               {
                   printf( "(%15.8f , %15.8f )\n", $1.lo, $1.hi );
               }
               DREG '=' dexp '\n'
               {
                   dreg[$1] = $3;
               }
               VREG '=' vexp '\n'
               {
                   vreg[$1] = $3;
               }
               error '\n'
               {
                   yyerrok;
               }
           }
           ;

dexp      : CONST
           {
               DREG
               {
                   $$ = dreg[$1]
               }
           }
           {
               dexp '+' dexp
               {
                   $$ = $1 + $3
               }
           }
           {
               dexp '-' dexp
               {
                   $$ = $1 - $3
               }
           }
           {
               dexp '*' dexp
               {
                   $$ = $1 * $3
               }
           }
           {
               dexp '/' dexp
               {
                   $$ = $1 / $3
               }
           }
           {
               '-' dexp    %prec UMINUS
               {
                   $$ = - $2
               }
           }
           {
               '(' dexp ')'
               {
                   $$ = $2
               }
           }
           ;

```



```

vexpp : dexp
      {
          $$hi = $$lo = $1;
      }
      { (' dexp ',' dexp ')
      {
          $$lo = $2;
          $$hi = $4;
          if( $$lo > $$hi )
          {
              printf( "interval out of order n" );
              YYERROR;
          }
      }
      }
      VREG
      {
          $$ = vreg[$1]
      }
      vexpp '+' vexpp
      {
          $$hi = $1hi + $3hi;
          $$lo = $1lo + $3lo
      }
      dexp '+' vexpp
      {
          $$hi = $1 + $3hi;
          $$lo = $1 + $3lo
      }
      vexpp '-' vexpp
      {
          $$hi = $1hi - $3lo;
          $$lo = $1lo - $3hi
      }
      dexp '-' vexpp
      {
          $$hi = $1 - $3lo;
          $$lo = $1 - $3hi
      }
      vexpp '*' vexpp
      {
          $$ = vmul( $1lo, $1hi, $3 )
      }
      dexp '*' vexpp
      {
          $$ = vmul( $1, $1, $3 )
      }
      vexpp '/' vexpp
      {
          if( dcheck( $3 ) ) YYERROR;
          $$ = vdiv( $1lo, $1hi, $3 )
      }
      dexp '/' vexpp
      {
          if( dcheck( $3 ) ) YYERROR;
          $$ = vdiv( $1, $1, $3 )
      }

```



```

    }
    | '-' vexp    %prec UMINUS
    {
        $$hi = -$2.lo; $$lo = -$2.hi
    }
    | '(' vexp ')'
    {
        $$ = $2
    }
    ;

%%

# define BSZ 50    /* buffer size for floating point numbers */

/* lexical analysis */

yylex( )
{
    register c;

    /* skip over blanks */
    while( (c=getchar( )) == ' ' )
        ;
    if( isupper( c ) )
    {
        yylval.ival = c - 'A';
        return( VREG );
    }
    if( islower( c ) )
    {
        yylval.ival = c - 'a';
        return( DREG );
    }

    /* gobble up digits, points, exponents */
    if( isdigit( c ) || c == '.' )
    {
        char buf[BSZ+1], *cp = buf;
        int dot = 0, exp = 0;

        for( ; (cp-buf)<BSZ ; ++cp,c=getchar( ) )
        {
            *cp = c;
            if( isdigit( c ) )
                continue;
            if( c == '.' )
            {
                if( dot++ || exp )
                    return( '.' ); /* will cause syntax error */
                continue;
            }
        }
    }
}

```



```

        if( c == 'e' )
        {
            if( exp++ )
                return( 'e' ); /* will cause syntax error */
            continue;
        }
        /* end of number */
        break;
    }
    *cp = '\0';
    if( (cp-buf) >= BSZ )
        printf( "constant too long: truncated\n" );
    else
        ungetc( c, stdin ); /* push back last char read */
    yylval.dval = atof( buf );
    return( CONST );
}
return( c );
}

```

## INTERVAL

hilo( a, b, c, d )

double a, b, c, d;

{

/\* returns the smallest interval containing a, b, c, and d \*/

/\* used by \*,-/ routines \*/

INTERVAL v;

if( a&gt;b )

{

v.hi = a;

v.lo = b;

}

else

{

v.hi = b;

v.lo = a;

}

if( c&gt;d )

{

if( c&gt;v.hi )

v.hi = c;

if( d&lt;v.lo )

v.lo = d;

}

else

{

if( d&gt;v.hi )

v.hi = d;

if( c&lt;v.lo )

v.lo = c;

}

return( v );



```
}  
  
INTERVAL vmul( a, b, v )  
    double a, b;  
    INTERVAL v;  
  
{  
    return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );  
}  
  
dcheck( v )  
    INTERVAL v;  
  
{  
    if( v.hi >= 0. && v.lo <= 0. )  
    {  
        printf( "divisor interval contains 0.\n" );  
        return( 1 );  
    }  
    return( 0 );  
}  
  
INTERVAL vdiv( a, b, v )  
    double a, b;  
    INTERVAL v;  
  
{  
    return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );  
}
```



## APPENDIX 10.4

## OLD FEATURES SUPPORTED BUT NOT ENCOURAGED

This appendix mentions synonyms and features which are supported for historical continuity but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotes.
2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or `_`, the type number of the literal is defined just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literals.

The use of multicharacter literals is likely to mislead those unfamiliar with `yacc` since it suggests that `yacc` is doing a job which must be actually done by the lexical analyzer.

3. Most places where `%` is legal, backslash `"\"` may be used. In particular, `\\` is the same as `%%`, `\left` the same as `%left`, etc.
4. There are a number of other synonyms:

- `%<` is the same as `%left`
- `%>` is the same as `%right`
- `%binary` and `%2` are the same as `%nonassoc`
- `%0` and `%term` are the same as `%token`
- `%=` is the same as `%prec`.

5. Actions may also have the form

`= { ... }`

and the curly braces can be dropped if the action is a single C language statement.

6. The C language code between `%{` and `%}` used to be permitted at the head of the rules section as well as in the declaration section.



NOTES



# Dokument Processing Guide UNIX System



Document Processing  
Group  
UNIX System

Trademarks:

|               |                          |
|---------------|--------------------------|
| MUNIX, CADMUS | for PCS                  |
| UNIX          | for Bell Laboratories    |
| PDP, VAX      | for DEC                  |
| TEKTRONIX     | for Tektronik, Inc.      |
| TELETYPE      | for Teletype Corporation |
| Versatec      | for Versatec Corporation |

Copyright 1984 by  
PCS GmbH, Pfälzer-Wald-Strasse 38, D-8000 München 90, tel. (089) 67804-0

The information contained herein is the property of PCS and shall neither be reproduced in whole or in part without PCS's prior written approval nor be implied to grant any license to make, use or sell equipment manufactured herewith.

PCS reserves the right to make changes without notice in the specifications and materials contained herein and shall not be responsible for any damages (including consequential) caused by reliance on the materials presented.



# DOCUMENT PROCESSING GUIDE

## UNIX SYSTEM

| CONTENTS                                      | PAGE |
|-----------------------------------------------|------|
| I. INTRODUCTION . . . . .                     | 13   |
| II. DOCUMENT PREPARATION . . . . .            | 15   |
| ADVANCED EDITING . . . . .                    | 15   |
| 1. Introduction . . . . .                     | 15   |
| 2. Special Characters . . . . .               | 15   |
| 2.1 Print and List Commands . . . . .         | 15   |
| 2.2 Substitute Command . . . . .              | 16   |
| 2.3 Undo Command . . . . .                    | 17   |
| 2.4 Metacharacters . . . . .                  | 17   |
| 3. Operating On Lines . . . . .               | 25   |
| 3.1 Substituting Newline Characters . . . . . | 25   |
| 3.2 Joining Lines . . . . .                   | 26   |
| 3.3 Rearranging Lines . . . . .               | 26   |
| 4. Line Addressing in Editor . . . . .        | 26   |
| 4.1 Address Arithmetic . . . . .              | 27   |
| 4.2 Repeated Searches . . . . .               | 28   |
| 4.3 Default Line Numbers . . . . .            | 28   |
| 4.4 Semicolon . . . . .                       | 30   |
| 4.5 Interrupting the Editor . . . . .         | 31   |
| 5. Global Commands . . . . .                  | 31   |



| CONTENTS                                                 | PAGE |
|----------------------------------------------------------|------|
| 5.1 Basic . . . . .                                      | 31   |
| 5.2 Multiline . . . . .                                  | 33   |
| 6. Cut and Paste . . . . .                               | 33   |
| 6.1 Command Functions . . . . .                          | 34   |
| 6.2 Text Editor Functions . . . . .                      | 36   |
| 6.3 Temporary Escape . . . . .                           | 39   |
| 7. Supporting Tools . . . . .                            | 39   |
| 7.1 Global Printing From a Set of Files (grep) . . . . . | 39   |
| 7.2 Editing Scripts . . . . .                            | 40   |
| STREAM EDITOR . . . . .                                  | 41   |
| 1. Introduction . . . . .                                | 41   |
| 2. Overall Operation . . . . .                           | 41   |
| 2.1 Command Line Flags . . . . .                         | 41   |
| 2.2 Order of Application of Editing Commands . . . . .   | 42   |
| 2.3 Pattern Space . . . . .                              | 42   |
| 2.4 Examples . . . . .                                   | 42   |
| 3. Selecting Lines for Editing . . . . .                 | 42   |
| 3.1 Line Number Addresses . . . . .                      | 42   |
| 3.2 Context Addresses . . . . .                          | 43   |
| 3.3 Number of Addresses . . . . .                        | 44   |
| 4. Functions . . . . .                                   | 44   |
| 4.1 Whole Line Oriented Functions Summary . . . . .      | 44   |
| 4.2 Substitute Function . . . . .                        | 46   |
| 4.3 Input/Output Functions . . . . .                     | 47   |
| 4.4 Multiple Input Line Functions . . . . .              | 48   |
| 4.5 Hold and Get Functions . . . . .                     | 49   |



| CONTENTS                                                                   | PAGE |
|----------------------------------------------------------------------------|------|
| 4.6 Flow of Control Functions . . . . .                                    | 49   |
| 4.7 Miscellaneous Functions . . . . .                                      | 50   |
| MISCELLANEOUS FACILITIES . . . . .                                         | 51   |
| III. FORMATTING FACILITIES . . . . .                                       | 53   |
| NROFF AND TROFF USER'S MANUAL . . . . .                                    | 53   |
| 1. Introduction . . . . .                                                  | 53   |
| 2. Usage . . . . .                                                         | 53   |
| 3. NROFF/TROFF Reference Manual . . . . .                                  | 57   |
| 3.1 General Explanation . . . . .                                          | 57   |
| 3.2 Font and Character Size Control . . . . .                              | 59   |
| 3.3 Page Control . . . . .                                                 | 60   |
| 3.4 Text Filling, Adjusting, and Centering . . . . .                       | 60   |
| 3.5 Vertical Spacing . . . . .                                             | 61   |
| 3.6 Line Length and Indenting . . . . .                                    | 61   |
| 3.7 Macros, Strings, Diversions, and Position Traps . . . . .              | 62   |
| 3.8 Number Registers . . . . .                                             | 64   |
| 3.9 Tabs, Leaders, and Fields . . . . .                                    | 65   |
| 3.10 Input/Output Conventions and Character Translations . . . . .         | 66   |
| 3.11 Local Horizontal/Vertical Motion and Width Function . . . . .         | 67   |
| 3.12 Overstrike, Zero-Width, Bracket, and Line Drawing Functions . . . . . | 68   |
| 3.13 Hyphenation . . . . .                                                 | 70   |
| 3.14 Three-Part Titles . . . . .                                           | 70   |
| 3.15 Output Line Numbering . . . . .                                       | 70   |
| 3.16 Conditional Acceptance of Input . . . . .                             | 70   |
| 3.17 Environment Switching . . . . .                                       | 71   |
| 3.18 Insertions From Standard Input . . . . .                              | 71   |



## CONTENTS

## PAGE

|                                                |    |
|------------------------------------------------|----|
| 3.19 Input/Output File Switching . . . . .     | 72 |
| 3.20 Miscellaneous . . . . .                   | 72 |
| 3.21 Output and Error Messages . . . . .       | 72 |
| 3.22 Compacted Macros . . . . .                | 72 |
| 4. TROFF Tutorial . . . . .                    | 74 |
| 4.1 Overview . . . . .                         | 74 |
| 4.2 Point Sizes and Line Spacing . . . . .     | 75 |
| 4.3 Fonts and Special Characters . . . . .     | 76 |
| 4.4 Indents and Line Lengths . . . . .         | 77 |
| 4.5 Tabs . . . . .                             | 78 |
| 4.6 Local Motions . . . . .                    | 79 |
| 4.7 Strings . . . . .                          | 81 |
| 4.8 Introduction to Macros . . . . .           | 82 |
| 4.9 Titles, Pages, and Numbering . . . . .     | 83 |
| 4.10 Number Registers and Arithmetic . . . . . | 85 |
| 4.11 Macros With Arguments . . . . .           | 87 |
| 4.12 Conditionals . . . . .                    | 89 |
| 4.13 Environments . . . . .                    | 90 |
| 4.14 Diversions . . . . .                      | 90 |
| 5. NROFF/TROFF Tutorial Examples . . . . .     | 91 |
| 5.1 Page Margins . . . . .                     | 91 |
| 5.2 Paragraphs and Headings . . . . .          | 93 |
| 5.3 Multiple Column Output . . . . .           | 94 |
| 5.4 Footnote Processing . . . . .              | 94 |
| 5.5 Last Page . . . . .                        | 96 |
| TABLE FORMATTING PROGRAM . . . . .             | 97 |



| CONTENTS                                         |                                                            | PAGE |
|--------------------------------------------------|------------------------------------------------------------|------|
| 1.                                               | Introduction . . . . .                                     | 97   |
| 2.                                               | Usage . . . . .                                            | 97   |
| 3.                                               | Input Commands . . . . .                                   | 98   |
| 3.1                                              | Global Options . . . . .                                   | 99   |
| 3.2                                              | Format Section . . . . .                                   | 99   |
| 3.3                                              | Data To Be Printed . . . . .                               | 102  |
| 4.                                               | Additional Command Lines . . . . .                         | 103  |
| 5.                                               | Examples . . . . .                                         | 103  |
| <b>MATHEMATICS TYPESETTING PROGRAM</b> . . . . . |                                                            | 105  |
| 1.                                               | Introduction . . . . .                                     | 105  |
| 2.                                               | Usage . . . . .                                            | 105  |
| 3.                                               | Language . . . . .                                         | 106  |
| 3.1                                              | Design . . . . .                                           | 106  |
| 3.2                                              | Structure . . . . .                                        | 107  |
| 3.3                                              | Mode of Operation . . . . .                                | 107  |
| 4.                                               | User's Guide . . . . .                                     | 107  |
| 4.1                                              | Delimiters . . . . .                                       | 107  |
| 4.2                                              | Spaces and New Lines . . . . .                             | 108  |
| 4.3                                              | Symbols, Special Names, and Greek Alphabet . . . . .       | 108  |
| 4.4                                              | Subscripts and Superscripts . . . . .                      | 109  |
| 4.5                                              | Braces . . . . .                                           | 111  |
| 4.6                                              | Fractions . . . . .                                        | 111  |
| 4.7                                              | Square Roots . . . . .                                     | 112  |
| 4.8                                              | Summations, Integrals, and Similar Constructions . . . . . | 112  |
| 4.9                                              | Size and Font Changes . . . . .                            | 113  |
| 4.10                                             | Diacritical Marks . . . . .                                | 114  |



| CONTENTS                                       | PAGE |
|------------------------------------------------|------|
| 4.11 Quoted Text . . . . .                     | 115  |
| 4.12 Aligning Equations . . . . .              | 115  |
| 4.13 Big Brackets . . . . .                    | 116  |
| 4.14 Piles . . . . .                           | 117  |
| 4.15 Matrices . . . . .                        | 118  |
| 4.16 In-Line Equations . . . . .               | 118  |
| 4.17 Defines . . . . .                         | 119  |
| 4.18 Local Motions . . . . .                   | 120  |
| 4.19 Precedence . . . . .                      | 120  |
| 5. Troubleshooting . . . . .                   | 120  |
| IV. MEMORANDUM MACROS . . . . .                | 157  |
| 1. Introduction . . . . .                      | 157  |
| 1.1 Purpose . . . . .                          | 157  |
| 1.2 Conventions . . . . .                      | 157  |
| 1.3 Document Structure . . . . .               | 157  |
| 1.4 Input Text Structure . . . . .             | 158  |
| 1.5 Definitions . . . . .                      | 158  |
| 2. Usage . . . . .                             | 159  |
| 2.1 The mm Command . . . . .                   | 159  |
| 2.2 The -cm or -mm Flag . . . . .              | 160  |
| 2.3 Typical Command Lines . . . . .            | 160  |
| 2.4 Parameters Set From Command Line . . . . . | 162  |
| 2.5 Omission of -cm or -mm Flag . . . . .      | 164  |
| 3. Formatting Concepts . . . . .               | 164  |
| 3.1 Basic Terms . . . . .                      | 164  |
| 3.2 Arguments and Double Quotes . . . . .      | 165  |



| CONTENTS                                                    | PAGE |
|-------------------------------------------------------------|------|
| 3.3 Unpaddable Spaces . . . . .                             | 165  |
| 3.4 Hyphenation . . . . .                                   | 166  |
| 3.5 Tabs . . . . .                                          | 166  |
| 3.6 BEL Character . . . . .                                 | 167  |
| 3.7 Bullets . . . . .                                       | 167  |
| 3.8 Dashes, Minus Signs, and Hyphens . . . . .              | 167  |
| 3.9 Trademark String . . . . .                              | 167  |
| 3.10 Use of Formatter Requests . . . . .                    | 168  |
| 4. Paragraphs and Headings . . . . .                        | 168  |
| 4.1 Paragraphs . . . . .                                    | 168  |
| 4.2 Numbered Headings . . . . .                             | 170  |
| 4.3 Unnumbered Headings . . . . .                           | 173  |
| 4.4 Headings and Table of Contents . . . . .                | 174  |
| 4.5 First-Level Headings and Page Numbering Style . . . . . | 174  |
| 4.6 User Exit Macros . . . . .                              | 174  |
| 4.7 Hints for Large Documents . . . . .                     | 176  |
| 5. Lists . . . . .                                          | 176  |
| 5.1 List Macros . . . . .                                   | 176  |
| 5.2 List-Begin Macro and Customized Lists . . . . .         | 182  |
| 5.3 User-Defined List Structures . . . . .                  | 183  |
| 6. Memorandum and Released-Paper Style Documents . . . . .  | 186  |
| 6.1 Sequence of Beginning Macros . . . . .                  | 186  |
| 6.2 Title . . . . .                                         | 186  |
| 6.3 Authors . . . . .                                       | 187  |
| 6.4 TM Numbers . . . . .                                    | 187  |
| 6.5 Abstract . . . . .                                      | 188  |



| CONTENTS                                                            | PAGE |
|---------------------------------------------------------------------|------|
| 6.6 Other Keywords . . . . .                                        | 188  |
| 6.7 Memorandum Types . . . . .                                      | 189  |
| 6.8 Date Changes . . . . .                                          | 190  |
| 6.9 Alternate First-Page Format . . . . .                           | 190  |
| 6.10 Example . . . . .                                              | 190  |
| 6.11 End of Memorandum Macros . . . . .                             | 191  |
| 6.12 One-Page Letter . . . . .                                      | 193  |
| 7. Displays . . . . .                                               | 193  |
| 7.1 Static Displays . . . . .                                       | 193  |
| 7.2 Floating Displays . . . . .                                     | 195  |
| 7.3 Tables . . . . .                                                | 196  |
| 7.4 Equations . . . . .                                             | 197  |
| 7.5 Figure, Table, Equation, and Exhibit Titles . . . . .           | 198  |
| 7.6 List of Figures, Tables, Equations, and Exhibits . . . . .      | 198  |
| 8. Footnotes . . . . .                                              | 199  |
| 8.1 Automatic Numbering of Footnotes . . . . .                      | 199  |
| 8.2 Delimiting Footnote Text . . . . .                              | 199  |
| 8.3 Format Style of Footnote Text . . . . .                         | 199  |
| 8.4 Spacing Between Footnote Entries . . . . .                      | 200  |
| 9. Page Headers and Footers . . . . .                               | 201  |
| 9.1 Default Headers and Footers . . . . .                           | 201  |
| 9.2 Header and Footer Macros . . . . .                              | 201  |
| 9.3 Default Header and Footer With Section-Page Numbering . . . . . | 202  |
| 9.4 Strings and Registers in Header and Footer Macros . . . . .     | 202  |
| 9.5 Header and Footer Example . . . . .                             | 203  |
| 9.6 Generalized Top-of-Page Processing . . . . .                    | 203  |



|     | CONTENTS                                               | PAGE |
|-----|--------------------------------------------------------|------|
|     | 9.7 Generalized Bottom-of-Page Processing . . . . .    | 204  |
|     | 9.8 Top and Bottom (Vertical) Margins . . . . .        | 204  |
|     | 9.9 Proprietary Marking . . . . .                      | 204  |
|     | 9.10 Private Documents . . . . .                       | 205  |
| 10. | Table of Contents and Cover Sheet . . . . .            | 205  |
|     | 10.1 Table of Contents . . . . .                       | 205  |
|     | 10.2 Cover Sheet . . . . .                             | 207  |
| 11. | References . . . . .                                   | 207  |
|     | 11.1 Automatic Numbering of References . . . . .       | 207  |
|     | 11.2 Delimiting Reference Text . . . . .               | 207  |
|     | 11.3 Subsequent References . . . . .                   | 208  |
|     | 11.4 Reference Page . . . . .                          | 208  |
| 12. | Miscellaneous Features . . . . .                       | 208  |
|     | 12.1 Bold, Italic, and Roman Fonts . . . . .           | 208  |
|     | 12.2 Justification of Right Margin . . . . .           | 209  |
|     | 12.3 SCCS Release Identification . . . . .             | 210  |
|     | 12.4 Two-Column Output . . . . .                       | 210  |
|     | 12.5 Column Headings for Two-Column Output . . . . .   | 211  |
|     | 12.6 Vertical Spacing . . . . .                        | 211  |
|     | 12.7 Skipping Pages . . . . .                          | 211  |
|     | 12.8 Forcing an Odd Page . . . . .                     | 212  |
|     | 12.9 Setting Point Size and Vertical Spacing . . . . . | 212  |
|     | 12.10 Producing Accents . . . . .                      | 213  |
|     | 12.11 Inserting Text Interactively . . . . .           | 213  |
| 13. | Errors and Debugging . . . . .                         | 214  |
|     | 13.1 Error Terminations . . . . .                      | 214  |



| CONTENTS                                                | PAGE |
|---------------------------------------------------------|------|
| 13.2 Disappearance of Output . . . . .                  | 214  |
| 14. Extending and Modifying MM Macros . . . . .         | 215  |
| 14.1 Naming Conventions . . . . .                       | 215  |
| 14.2 Sample Extensions . . . . .                        | 216  |
| 15. Summary . . . . .                                   | 218  |
| V. VIEWGRAPHS AND SLIDES MACROS . . . . .               | 237  |
| 1. Introduction . . . . .                               | 237  |
| 2. Examples . . . . .                                   | 237  |
| 2.1 Trivial Example . . . . .                           | 237  |
| 2.2 Less Trivial Example . . . . .                      | 238  |
| 2.3 Other Examples . . . . .                            | 238  |
| 3. Macros . . . . .                                     | 242  |
| 3.1 Foil-Start Macros . . . . .                         | 242  |
| 3.2 Level Macros . . . . .                              | 243  |
| 3.3 Titles . . . . .                                    | 245  |
| 3.4 Global Indents . . . . .                            | 245  |
| 3.5 Point Sizes and Line Lengths . . . . .              | 245  |
| 3.6 Default Fonts . . . . .                             | 245  |
| 3.7 Default Vertical Space . . . . .                    | 246  |
| 3.8 Underlining . . . . .                               | 246  |
| 3.9 Synonyms . . . . .                                  | 246  |
| 3.10 Breaks . . . . .                                   | 246  |
| 3.11 Text Filling, Adjusting, and Hyphenation . . . . . | 247  |
| 4. The troff Preprocessors . . . . .                    | 247  |
| 4.1 Tables . . . . .                                    | 247  |
| 4.2 Mathematical Expressions . . . . .                  | 247  |



|    | CONTENTS                                          | PAGE |
|----|---------------------------------------------------|------|
|    | 4.3 Constant-Width Program Examples . . . . .     | 247  |
| 5. | The Finished Product . . . . .                    | 247  |
|    | 5.1 Phototypesetter Output . . . . .              | 247  |
|    | 5.2 Output Approximation on a Terminal . . . . .  | 248  |
|    | 5.3 Making Actual Viewgraphs and Slides . . . . . | 248  |
| 6. | Suggestions For Use . . . . .                     | 248  |
| 7. | Warnings . . . . .                                | 250  |
|    | 7.1 Use of troff Formatter Requests . . . . .     | 250  |
|    | 7.2 Reserved Names . . . . .                      | 250  |
|    | 7.3 Miscellaneous . . . . .                       | 250  |
| 8. | Dimensional Details . . . . .                     | 250  |



1049

1050

1051

1052

1053

1054

1055

1056

1057

1058

1059

1060

1061

1062

1063

1064

1065

1066

1067

1068

1069

1070

1071

1072

1073

1074

1075

1076

1077

1078

1079

1080



## I. INTRODUCTION

An important feature of the UNIX operating system is to provide a method of document preparation and generation. The Document Preparation section (Section II) contains three parts:

- Advanced Editing
- Stream Editor
- Miscellaneous Facilities.

The Advanced Editing part describes text editing functions which include special characters, line addressing, global commands, commands for cut and paste operations, and text editor-based programs. The Stream Editor part describes the noninteractive context editor, and the Miscellaneous Facilities part outlines some other facilities that aid in document preparation.

The Formatting Facilities section (Section III) contains three parts:

- NROFF and TROFF User's Manual
- Table Formatting Program
- Mathematics Typesetting Program.

The NROFF and TROFF User's Manual part presents information to enable the user to do simple formatting tasks and to make incremental changes to existing packages of **troff** formatter commands. The UNIX operating system formatter, **nroff**, is identical to the **troff** formatter in most respects. The Table Formatting Program part describes the **tbl** program and the input commands used to generate documents that contain tables. The Mathematics Typesetting Program part describes the program usage and language for obtaining text with mathematical expressions. The language interfaces directly with the **troff** processor so mathematical expressions can be embedded in the running text of a manuscript and the entire document produced in one process.

The Memorandum Macros section (Section IV) is a user's guide and reference manual for the Memorandum Macros (MM). These macros provide a general purpose package of text formatting macros used with the **nroff** and **troff** formatters. The macros provide users of the UNIX operating system a unified, consistent, and flexible tool for producing many common types of documents. Although the UNIX operating system provides other macro packages for various specialized formats, MM is the standard general purpose macro package for most documents such as letters, reports, technical memoranda, released papers, manuals, books, design proposals, and user guides. Uses of MM range from single-page letters to documents of several hundred pages in length.

The Viewgraphs and Slides Macros section (Section V) describes the MV package of macros. These macros provide users a method of preparing viewgraphs and slides using the **troff** formatter. Included is a discussion on the use of the macros and a philosophy of how a viewgraph or slide should appear.







## II. DOCUMENT PREPARATION

### ADVANCED EDITING

#### 1. Introduction

The advanced editing part is meant to help UNIX operating system users (secretaries, typists, programmers, etc.) make effective use of facilities for preparing and editing documents, text, programs, files, etc. It provides explanations and examples of:

- special characters, line addressing, and global commands in the text editor (`ed`)
- commands for "cut and paste" operations on files and parts of files, including `mv`, `cp`, `cat`, and `rm` commands, and `r`, `w`, `m`, and `t` commands of the text editor
- editing scripts and text editor-based programs like `grep` and `sed`.

Although this document is written for nonprogrammers, new UNIX operating system users with any background should find helpful hints on how to get their jobs done more easily. The UNIX operating system provides effective tools for text editing, but that by itself is no guarantee that everyone will automatically make the most effective use of them. In particular, users who are not computer specialists (typists, secretaries, casual users) often use the UNIX operating system less effectively than they could. The reader should be familiar with the material in the "Basics For Beginners" section of the User's Guide—UNIX Operating System before using the text editor. Further information on all commands discussed here can be found in the User's Manual—UNIX Operating System.

Examples are based on experience and observations of users and the difficulties encountered. Topics covered include special characters in searches and substitute commands, line addressing, the global commands, and line moving and copying. There are also brief discussions on the effective use of related tools, e.g., those for file manipulation and those based on `ed`.

The next paragraphs discuss shortcuts and labor-saving devices. Not all will be instantly useful (some will) and others should provide ideas for future use. Until these things are used to build confidence, they will remain theoretical knowledge.

**Note:** A document like this should provide ideas about what to try. There is only one way to learn to use something, and that is to use it. Reading a description is no substitute for hands-on use.

#### 2. Special Characters

The `ed` program is the primary interface to the system, so it is worthwhile to know how to get the most out of it with the least effort.

##### 2.1 Print and List Commands

Two commands are provided for printing contents of lines being edited. Most users are familiar with the print command (`p`) in combinations like

`1,$p`

to print all lines that are being edited, or

`s/abc/def/p`

to change "abc" to "def" on the current line and to print the results. Less familiar is the list command (`l`) which gives slightly more information than `p`. In particular, `l` makes visible characters that are normally invisible,



such as tabs and backspaces. If a line listed contains some of these, `l` will print each tab as ">" and each backspace as "<". This makes it easier to correct typing mistakes that insert extra spaces adjacent to tabs or a backspace followed by a space.

The `l` command also "folds" long lines for printing. Any line that exceeds 72 characters is printed on multiple lines. Each printed line except the last is automatically terminated by a backslash (\) to indicate that the line was folded. A "\$" character is appended to the real end of line. This is useful for printing long lines on terminals having output line capability of only 72 characters per line.

Occasionally, the `l` command will print in a line a string of numbers preceded by a backslash, such as \07 or \16. These combinations are used to make visible characters that normally do not print, e.g., form feed, vertical tab, or bell. Each such combination is interpreted as a single character. When such characters are detected, they may have surprising meanings when printed on some terminals. Often their presence means that a finger slipped while typing.

## 2.2 Substitute Command

The substitute command (`s`) is used for changing the contents of individual lines. It is probably the most complex and effective of any `ed` command.

The meaning of a trailing global command after a substitute command is illustrated in the next two commands:

```
s/this/that/
```

and

```
s/this/that/g
```

The first form replaces the first "this" on the line with "that". If there is more than one occurrence of "this" on the line, the second form (with the trailing `g`) changes all of them. Either of the two forms of the `s` command can be followed by `p` or `l` to print or list the contents of the line:

```
s/this/that/p  
s/this/that/l  
s/this/that/gp  
s/this/that/gl
```

All are legal and have slightly different meanings.

An `s` command can be preceded by one or two line numbers to specify that the substitution is to take place on a group of lines specified by the line numbers. Thus:

```
1,$s/mispell/misspell/
```

changes the first occurrence of "mispell" to "misspell" on every line of the file. The following command changes every occurrence in every line:

```
1,$s/mispell/misspell/g
```

By adding a `p` or `l` to the end of any of these substitute commands, only the last line that was changed will be printed, not all lines. How to print all the lines that were changed is described later.



Any character can be used to delimit pieces of an **s** command. There is nothing sacred about slashes (but slashes must be used for context searching). For instance, for a line that contains a lot of slashes already, e.g:

```
//exec //sys.fort.go //etc...
```

a colon could be used as the delimiter. To delete all the slashes, the command is

```
s/:::g
```

### 2.3 Undo Command

Occasionally, an erroneous substitution will be made in a line. The undo command (**u**) negates the last command so that data is restored to its previous state. This command is useful after executing a global command if it is discovered the command did things that are undesirable.

### 2.4 Metacharacters

When using **ed**, certain characters have unexpected meanings when they occur in the left side of a substitute command or in a search for a particular line. These are called "metacharacters" which are:

- Period
- Backslash
- Dollar Sign
- Circumflex
- Star
- Brackets
- Ampersand.

Even though metacharacters are discussed separately in the following text, they can be combined. An example is given in the paragraph on "Circumflex" (2.4.4).

#### 2.4.1 Period

The period (.) on the left side of a substitute command or in a search with **/.../**, stands for any single character. Thus the search

```
/x.y/
```

finds any line where "x" and "y" occur separated by a single character, as in

```
x+y  
x-y  
x<sp>y  
x.y
```

The **<sp>** stands for a space whenever needed to make it visible.

Since the period matches any single character, a way to deal with the "invisible" characters printed by **l** is available. For instance, if there is a line that when printed with the **l** command, appears as



... th\07is ...

and it is desired to get rid of the "\07" (the bell character), the most obvious solution is to try

```
s/\07//
```

This will fail. The brute force solution, which is to retype the entire line, is a reasonable tactic if the line in question is not too long. However, for a very long line, retyping could result in additional errors. Since "\07" really represents a single character, the command

```
s/.this/this/
```

gets the job done. The period matches the mysterious character between the "h" and the "i", whatever it is.

Since the period matches any single character, the command

```
s/./,/
```

converts the first character on a line into a ",".

As is true of many characters in **ed**, the period has several meanings depending on its context. This line shows all three:

```
.s/././
```

- The first period is the line number of the line being edited, which is called "dot".
- The second period is a metacharacter that matches any single character on that line (in this instance the first character of the line).
- The third period is the only one that really is an honest literal period. On the right side of a substitution, the period is not special.

#### 2.4.2 Backslash

Since a period means "any character", the question arises of what to do when a period is really needed. For example, to convert the line:

Now is the time.

into

Now is the time?

the backslash (\) is used. A backslash turns off any special meaning that the next character might have. In particular, \. converts the period from a "match anything" into a "match the period" statement. The \. pair of characters is considered by **ed** to be a single literal period. To replace the period with a question mark, the following command is used:

```
s/\./?/
```

The backslash can also be used when searching for lines that contain a special character. If a search is made to look for a line that contains

```
.PP
```



the search

```
/PP/
```

is not adequate. It will find a line like

```
THE APPLICATION OF ...
```

The period matches the letter "A". But if the command

```
\.PP/
```

is used, only the lines that contain ".PP" are found.

The backslash can also be used to turn off special meanings for characters other than the period. For example, to find a line that contains a backslash, the search

```
\/
```

will not work because the \ is not a literal backslash, but instead means that the second / no longer delimits the search. A search can be made for a literal backslash by preceding a backslash with another \:

```
\\
```

Similarly, searches can be made for a slash (/) with

```
\/
```

The backslash turns off the meaning of the immediately following / so that it does not terminate /.../ construction prematurely.

Some substitute commands, each of which will convert the line

```
\x\y
```

into the line

```
\xy
```

are

```
s/\\\/  
s/x../x/  
s/..y/y/
```

The user's erase character and the line kill character (# and @ by default) must also be used with a backslash to turn off their special meaning. This is a feature of the UNIX operating system. When adding text with append (a), insert (i), or change (c) commands, the backslash is special only for the erase and line kill characters, and only one backslash should be used for each one needed.

#### 2.4.3 Dollar Sign

In the left side of a substitute command or in a search command the dollar sign (\$) stands for "the end of line". The word "time" is added to the end of the following phrase.



Now is the  
with the following command:

```
s/$/<sp>time/
```

The result is

Now is the time

A space is needed before "time" in the substitute command, otherwise, the following will be printed.

Now is thetime

The second comma in the following line can be replaced with a period without altering the first.

Now is the time, for all good men,

The needed command is

```
s/,$/./
```

The \$ provides context to indicate which specific comma. Without it the s command would operate on the first comma to produce

Now is the time. for all good men,

To convert

Now is the time.

into

Now is the time?

that was previously done with the backslash, the following command is used:

```
s/.$/?/
```

The \$ has multiple meanings depending on context. In the line

```
$s/$/$/
```

- The first \$ refers to the last line of the file.
- The second \$ refers to the end of the line.
- The third \$ is a literal dollar sign to be added to that line.

#### 2.4.4 Circumflex

The circumflex (^), alias "hat" or "caret", stands for the beginning of the line. For example, if a search is made for a line that begins with "the", the command

```
/the/
```



will in all likelihood find several lines that contain "the" before arriving at the line that was wanted. But the command

```
/^the/
```

narrows the context, and thus arrives at the desired line more easily.

The other use of ^ is to enable context to be inserted at the beginning of a line.

```
s/^/<sp>/
```

places a space at the beginning of the current line.

Metacharacters can be combined. For example, to search for a line that contains only the characters

```
.PP
```

the command

```
/^\.PP$/
```

can be used.

#### 2.4.5 Star

The star (\*) is useful to replace all spaces between *x* and *y* with a single space, as in the following example:

```
text x      y text
```

where *text* stands for lots of text, and there are an indeterminate number of spaces between *x* and *y*. The line is too long to retype, and there are too many spaces to count.

A regular expression (typically a single character) followed by a star stands for as many consecutive occurrences of that regular expression as possible. To refer to all the spaces at once, the following command is used:

```
s/x<sp>*y/x<sp>y/
```

The construction *<sp>\** means "as many spaces as possible". Thus *x<sp>\*y* means: "an *x*, followed by as many spaces as possible, and then a *y*".

The star can be used with any character, not just space. If the original example was

```
text x-----y text
```

then all "-" characters can be replaced by a single space with the command

```
s/x-*y/x<sp>y/
```

If the original line was

```
text x.....y text
```

and if the following command was typed:

```
s/x.*y/x<sp>y/
```



what happens depends upon the occurrence of other x's or y's on the line. If there are no other x's or y's, then everything works, but it is blind luck, not good management. Since a period matches any single character, then `.*` matches as many single characters as possible. Unless the user is careful the star can eat up a lot more of the line than expected. If the line was

```
text x text x.....y text y text
```

then the command will take everything from the first "x" to the last "y", which, in this example, is undoubtedly more than wanted. The proper way is to turn off the special meaning of period with `\.`:

```
s/x\.*y/x<sp>y/
```

Now everything works since `\.*` means "as many periods as possible".

There are times when the pattern `.*` is exactly what is wanted. For example, to change

```
Now is the time for all good men ...
```

into

```
Now is the time.
```

the following deletes everything after the word "time":

```
s/<sp>for.*//
```

There are a couple of additional pitfalls associated with `*` to be aware of. Most notable is that "as many as possible" means zero or more. The fact that zero is a legitimate possibility is sometimes rather surprising. For example, if this line contained

```
text xy text x          y text
```

and the command is

```
s/x<sp>*y/x<sp>y/
```

the first "xy" matches this pattern, for it consists of an "x", zero spaces, and a "y". The result is that the substitute acts on the first "xy" and does not touch the later one that actually contains some intervening spaces.

The proper way is to specify a pattern like

```
/x<sp><sp>*y/
```

which says "an x, a space, as many more spaces as possible, and then a y" (in other words, one or more spaces).

The other startling behavior of `*` is also related to the zero being a legitimate number of occurrences of something followed by a star. The command

```
s/x*/y/g
```

when applied to the line

```
abcdef
```



produces

yaybycydyeyfy

which is almost certainly not what was intended. The reason for this behavior is that zero is a legal number of matches, and there is no "x" at the beginning of the line (so that gets converted into a "y"), nor between the "a" and the "b" (so that gets converted into a "y"), etc. The following command:

s/xx\*/y/g

where "xx\*" is "one or more x's", when applied to the line

abcdefghijkl

produces

abcedfyghi

#### 2.4.6 Brackets

Should a number that appears at the beginning of all lines of a file need to be deleted, a first thought might be to perform a series of commands like:

```
1,$s/^1*//
1,$s/^2*//
1,$s/^3*//
```

This is going to take forever if the numbers are long. Unless it is desired to repeat the commands over and over until finally all numbers are gone, the digits can be deleted on one pass. This is the purpose of brackets ([ ]).

The construction

[0123456789]

matches any single digit. The whole thing is called a "character class". With a character class, the job is easy. The pattern "[0123456789]\*" matches zero or more digits (an entire number), so

```
1,$s/^ [0123456789]*//
```

deletes all digits from the beginning of all lines.

Any characters can appear within a character class; and just to confuse the issue, there are essentially no special characters inside the brackets. Even the backslash does not have a special meaning. The following command searches for special characters within the brackets:

```
/[.\$^[]/
```

Within a character class, the [ is not special. To get a ] into a character class, it should be placed as the first character in the class. For example:

```
/[ ]\.$[ ]/
```

It is a nuisance to have to spell out the digits. They can be abbreviated as [0-9]; similarly, [a-z] stands for the lowercase letters and [A-Z] for uppercase letters.



The user can specify a character class that means "none of the following characters". This is done by beginning the class with a circumflex.

```
[^0-9]
```

which stands for "any character except a digit". The following search finds the first line that does not begin with a tab or space:

```
/^[^(space)(tab)]/
```

Within a character class, the circumflex has a special meaning only if it occurs at the beginning. For example:

```
/^[^^]/
```

finds a line that does not begin with a circumflex.

#### 2.4.7 Ampersand

The ampersand (&) is used primarily to save typing. For example, if the following is the original line:

Now is the time

and it needs to be

Now is the best time

the command

```
s/the/the best/
```

can be used, but it is unnecessary to repeat the "the". The & is used to eliminate the repetition. On the right-hand side of a substitute command, the ampersand means "whatever was just matched", so in the command

```
s/the/& best/
```

the & represents "the". This is not much of a saving if the text matched is just "the"; but if it is something long or complicated or if it is something (such as .\*) which matches a lot of text, the & can save some tedious typing. There is also much less chance of making a typing error in the replacement text. For example, to parenthesize a line, regardless of its length:

```
s/.*/(&)/
```

The ampersand can occur more than once on the right side.

```
s/the/& best and & worst/
```

makes the original line

Now is the best and the worst time

and

```
s/.*/&? &!!/
```



converts the original line into

Now is the time? Now is the time!!

To get a literal ampersand, the backslash is used to turn off the special meaning.

s/ampersand/\&/

converts the word into the symbol. The & is not special on the left-hand side of a substitute, only on the right.

### 3. Operating On Lines

#### 3.1 Substituting Newline Characters

The **ed** program provides a facility for splitting a single line into two or more lines by substituting in a newline character. If a line is unmanageably long because of editing or merely because of the way it is typed, it can be divided as follows:

text xy text

can be broken between the "x" and the "y" with the following substitute command:

s/xy/x\  
y/

This is actually a single command although it is typed on two lines. Bearing in mind that \ turns off special meanings, it seems relatively intuitive that a \ at the end of a line would make the newline character there no longer special.

A single line can be made into several lines with this same mechanism. The word "very" in the following example can be put on a separate line preceded with the **nroff** formatter underline command (.ul):

text a very big text

The commands

s/<sp>very<sp>/\  
.ul\  
very\  
/

convert the line into four shorter lines:

text a  
.ul  
very  
big text

The word "very" is preceded by the line containing the ".ul" and spaces around "very" are eliminated at the same time.

When a new line is substituted in, dot is left pointing at the last line created.



### 3.2 Joining Lines

Lines may be joined together with the `j` command. Given the lines

```
Now is  
<sp>the time
```

and if `dot` is set to the first line, then the `j` command joins them together. No spaces are added, which is why a space is shown at the beginning of the second line.

All by itself, a `j` command joins `dot` to `dot+1`. Any contiguous set of lines can be joined by specifying the starting and ending line numbers. For example:

```
1,$jp
```

joins all the lines into a big one and prints it.

### 3.3 Rearranging Lines

The `&` metacharacter stands for whatever was matched by the left side of an `s` command. Similarly, several pieces can be captured of what was matched; the only difference is it must be specified on the left side just what pieces the user is interested in. For instance, if there is a file of lines that consist of names in the form

```
Smith, A. B.  
Jones, C.
```

etc., and it was intended to have the initials to precede the name, as in:

```
A. B. Smith  
C. Jones
```

it is possible to do this with a series of tedious and error-prone editing commands. The alternative is to "tag" the pieces of the pattern (in this case, the last name and the initials) and then rearrange the pieces. On the left side of a substitution if part of the pattern is enclosed between `\(` and `\)`, whatever matched that part is remembered and available for use on the right side. On the right side, the symbol `\1` refers to whatever matched the first `\(...\)` pair, `\2` to the second `\(...\)` pair, etc.

The command

```
1,$s/^\\([^\,]*)\\),<sp>*\\(.*)\\)/\2<sp>\1/
```

although hard to read, does the job. The first `"\\(...\\)"` matches the last name, which is any string up to the comma; this is referred to on the right side with `"\1"`. The second `"\\(...\\)"` is whatever follows the comma and any spaces and is referred to as `"\2"`.

With any complicated editing sequence, it is foolhardy to run it and hope. Global commands (see paragraphs 5.1 and 5.2) provide a way to print those lines affected by the substitute command.

## 4. Line Addressing in Editor

Line addressing in `ed` specifies the lines to be affected by editing commands. Previous constructions like

```
1,$s/x/y/
```

were used to specify a change on all lines. Most users are familiar with using a single newline character (or return) to print the next line and with



/string/

to find a line that contains "string". Less familiar is the use of

?string?

to scan backwards for the previous occurrence of "string". This is handy when the user realizes that the string to be operated on is back up the page (file) from the current line being edited.

The slash and question mark are the only characters that can be used to delimit a context search. Essentially, any character can be used as a delimiter in a substitute command.

#### 4.1 Address Arithmetic

The next step is to combine the line numbers like ., \$, /.../, and ?...? with + and -. Thus:

\$-1

is a command to print the next to last line of the current file (i.e., one line before line \$). For example:

\$-5,\$p

prints the last six lines. If there are not six lines, an error message will be indicated.

As another example:

.-3, +3p

prints from three lines before the current line to three lines after, thus printing a bit of context. The + can be omitted:

.-3,3p

is identical in meaning.

Another area in which to save typing effort in specifying lines is by using - and + as line numbers by themselves. For instance, a

-

by itself is a command to move back up one line in the file. Several minus signs can be strung together to move back up that many lines:

---

moves up three lines, as does "-3". Thus:

-3, +3p

is also identical to the examples above.

Since "-" is shorter than "-1", constructions like

-,s/bad/good/



are useful. This changes the first occurrence of "bad" to "good" on both the previous line and the current line.

The + and - can be used in combination with searches using /.../, ?...?, and \$. The search

```
/string/--
```

finds the line containing "string" and positions dot two lines before it.

#### 4.2 Repeated Searches

When the search command is

```
/horrible string/
```

and when the line is printed, it is discovered that it is not the horrible string that was wanted. It is necessary to repeat the search again, but it is not necessary to retype it. The construction

```
//
```

is a shorthand for "the string that was previously searched for", whatever it was. This can be repeated as many times as necessary. This also applies to the backwards search

```
??
```

which searches for the same string but in the reverse direction.

Not only can the search be repeated, but the // construction can be used on the left side of a substitute command to mean "the most recent pattern":

```
/horrible string/
--- ed prints line with "horrible string"
s//good/p
```

To go backwards and change a line, the following command is used:

```
??s//good/
```

Of course, the & on the right-hand side of a substitute can still be used to stand for whatever got matched:

```
//s//&<sp>&/p
```

finds the next occurrence of whatever was searched for last, replaces it by two copies of itself, and then prints the line just to verify that it worked.

#### 4.3 Default Line Numbers

One of the most effective ways to speed editing is by knowing which lines are affected by a command with no address and where dot will be positioned when a command finishes. Editing without specifying unnecessary line numbers can save a lot of typing. As the most obvious example, the search command

```
/string/
```

puts dot at the next line that contains "string". No address is required with commands like:

- s to make a substitution on the line



- p to print the line
- l to list the line
- d to delete the line
- a to append text after the line
- c to change the line
- i to insert text before the line.

If there was no "string", dot stays on the line where it was. This is also true if it was sitting on the only "string" when the command was issued. The same rules hold for searches that use ?...?; the only difference is direction of search.

The delete command (d) leaves dot at the line following the last deleted line. However, dot points to the new last line when the last line is deleted.

Line-changing commands a, c, and i affect (by default) the current line if no line number is specified. They behave identically in one respect—after appending, changing, or inserting, dot points at the last line entered. For example, the following can be done without specifying any line number for the substitute command or for the second append command:

```
a
  --- text
  --- botch      (minor error)
.
s/botch/correct/  (fix botched line)
a
  --- more text
```

The following overwrites the major error and permits continuation of entering information:

```
a
  --- text
  --- horrible botch  (major error)
.
c
  --- fixed up line   (replace entire line)
  --- more text
```

The read command (r) will read a file into the text being edited, either at the end if no address is given or after the specified line if an address is given. In either case, dot points at the last line read in. The Or command can be used to read in a file at the beginning of the text, and the Oa or li commands can be used to start adding text at the beginning.

The write command (w) writes out the entire file. If the command is preceded by one line number, that line is written. Preceding the command by two line numbers causes a range of lines to be written. The w command does not change dot, therefore, the current line remains the same regardless of what lines are written. This is true even if a command like

```
/^\.AB/,/^\.AE/w abstract
```

is made, which involves a context search. Since the w command is easy to use, the text being edited should be saved regularly just in case the system crashes or a file being edited is clobbered.



The command with the least intuitive behavior is the `s` command. The dot remains at the last line that was changed. If there were no changes, then dot is unchanged. To illustrate, if there are three lines in the buffer and dot is sitting on the middle one

```
x1
x2
x3
```

the command

```
-,+s/x/y/p
```

prints the third line, which is the last one changed. But if the three lines had been

```
x1
y2
y3
```

and the same command issued while dot pointed at the second line, then the result would be to change and print only the first line and that is where dot would be set.

#### 4.4 Semicolon

Searches with `/.../` and `?...?` start at the current line and move forward or backward, respectively, until they either find the pattern or get back to the current line. Sometimes this is not what is wanted. Suppose, for example, that the buffer contains lines like

```
.
.
.
ab
.
.
bc
.
.
.
```

Starting at line 1, one would expect that the command

```
/a/,/b/p
```

prints all the lines from the "ab" to the "bc", inclusive. This is not what happens. Both searches (for "a" and for "b") start from the same point, and thus they both find the line that contains "ab". The result is to print a single line. If there had been a line with a "b" in it before the "ab" line, then the print command would be in error since the second line number would be less than the first; and it is illegal to try to print lines in reverse order. This is because the comma separator for line numbers does not set dot while each address is processed. Each search starts from the same place.

In `ed`, the semicolon (`;`) can be used just like the comma with the single difference being that use of a semicolon forces dot to be set at that point while line numbers are being evaluated. In effect, the semicolon "moves" dot. Thus, in the example above, the command

```
/a;/b/p
```



prints the range of lines from "ab" to "bc" because after the "a" is found dot is set to that line, and then "b" is searched for starting beyond that line. This property is most often useful in a very simple situation. If the need is to find the second occurrence of "string", then the commands

```
/string/  
//
```

print the first occurrence as well as the second. The command

```
/string;/
```

finds the first occurrence of "string" and sets dot there. Then it finds the second occurrence and prints only that line.

Searching for the second previous occurrence of "string", as in

```
?string?;??
```

is similar. Printing the third, fourth, etc. occurrence in either direction is left as an exercise.

When searching for the first occurrence of a character string in a file where dot is positioned at an arbitrary place within the file, the command

```
1;/string/
```

will fail if "string" occurs on line 1. It is possible to use the command

```
0;/string/
```

(one of the few places where 0 is a legal line number) to start the search at line 1.

#### 4.5 Interrupting the Editor

If the user interrupts `ed` while performing a command (by depressing the BREAK key, the INTERRUPT key, or the user interrupt character [RUB OUT or DEL CHAR keys by default]), the file is put back together again. The file state is restored as much as possible to what it was before the command began. Naturally, some changes are irrevocable. If the file is being read from or written into, substitutions are being made, or lines are being deleted, these will be stopped in some clean but unpredictable state in the middle of the command execution (which is why it is not usually wise to stop them). Dot may or may not be changed.

Printing is more clear cut. Dot is not changed until the printing is done. Thus, if a user interrupts `ed` while some printing is being done, dot is not sitting on the last printed line or even near it. Dot is returned to where it was when the `p` command was started.

### 5. Global Commands

#### 5.1 Basic

Global commands (`g` and `v`) are used to perform one or more editing commands on all lines of a file. The `g` command operates on those lines that contain a specified string. As the simplest example, the command

```
g/THIS/p
```

prints all lines that contain the string "THIS". The string that goes between the slashes can be anything that could be used in a line search or in a substitute command; exactly the same rules and limitations apply. As another example:

```
g/^\. /p
```



prints all lines that begin with a period.

The **v** command (there is no mnemonic significance to the letter "v") is identical to **g**, except that it operates on those lines that do not contain an occurrence of the string. So

```
v/^\. /p
```

prints the lines that do not begin with a period.

The command that follows **g** or **v** can be almost any command. For example:

```
g/^\. /d
```

deletes all lines that begin with a period, and

```
g/^$/d
```

deletes all empty (blank) lines.

Probably the most useful command that can follow a global command is the substitute command since this can be used to make a change and print each affected line for verification. For example, to change the word "This" to "THIS" everywhere in a file and verify that it really worked, the command is

```
g/This/s//THIS/gp
```

The use of **//** in the substitute command means "the previous pattern", in this case, "This". The **p** command is done on every line that matches the pattern, not just those on which a substitution took place.

Global commands operate by making two passes over the file. On the first pass, all lines that match the pattern are marked. On the second pass, each marked line in turn is examined, dot is set to that line, and the command executed. This means that it is possible for the command that follows a **g** or **v** to use addresses, set dot, etc., quite freely. For example:

```
g/^\.PP/+
```

prints the line that follows each ".PP" command (the signal for a new paragraph in some formatting packages). The **+** means "one line past dot", and

```
g/topic/?^\.SH?1
```

searches for each line that contains "topic", scans backwards until it finds a line that begins ".SH" (a section heading) and prints the line that follows, thus showing the section headings under which "topic" is mentioned. Finally:

```
g/^\.EQ/+/^\.EN/-p
```

prints all the lines that lie between lines beginning with the ".EQ" and ".EN" formatting commands.

The **g** and **v** commands can also be preceded by line numbers, in which case the lines searched are only those in the range specified.



## 5.2 Multiline

It is possible to do more than one command under the control of a global command although the syntax for expressing the operation is not especially natural or easy. As an example, suppose the task is to change "x" to "y" and "a" to "b" on all lines that contain "string". Then:

```
g/string/s/x/y/\
s/a/b/
```

is sufficient. The backslash signals the **g** command that the set of commands continues on the next line. It terminates on the first line that does not end with \. A substitute command can not be used to insert a newline character within a **g** command.

The command

```
g/x/s//y/\
s/a/b/
```

does not work as expected. The remembered pattern is the last pattern that was actually executed, so sometimes it will be "x" (as expected) and sometimes it will be "a" (not expected). The desired pattern should be spelled out:

```
g/x/s/x/y/\
s/a/b/
```

It is also possible to execute **a**, **c**, and **i** commands (append, change, and insert) under a global command. As with other multiline constructions, all that is needed is to add a \ at the end of each line except the last. Thus to add a **.nf** and **.sp** command before each ".EQ" line, the following is typed:

```
g/^\.EQ/i\
.nf\
.sp
```

There is no need for a final line containing a period to terminate the **i** command unless there are further commands being done under the global. On the other hand, it does no harm to put it in.

It is good practice, after each global command, to check that the command did only what was desired. Surprises sometimes happen. When they do occur, the **u** command (undo) is useful to negate what was done by the last command.

## 6. Cut and Paste

One editing area in which nonprogrammers do not seem confident is the "cut and paste" operations. There are two areas in which the operations can be performed. Using the UNIX operating system command functions, the following can be done:

- Changing the name of a file
- Making a copy of a file somewhere else
- Combining files
- Removing a file.

The text editor (**ed**) function performs the following operations.



- Inserting one file in the middle of another
- Splitting a file into pieces
- Moving a few lines from one place to another in a file
- Copying lines.

Most of these operations are actually quite easy if the task is defined and precautions are taken when entering the commands.

### 6.1 Command Functions

Changing file names, making copies of files, combining files, and removing files are handled with the UNIX operating system commands.

#### 6.1.1 Changing Name of Files

If there is a file named *oldname* and if it needs to be renamed to *newname*, the move command (**mv**) will do the job. It moves the file from one name to another (the target file), for example

```
mv oldname newname
```

**Note:** If there is already a file with the new name, its contents will be overwritten with information from the other (*oldname*) file. The one exception is that a file cannot be moved to itself; therefore, the following command is illegal.

```
mv oldname oldname
```

#### 6.1.2 Copying Files

Sometimes a copy of a file is needed while retaining the original file. This might be because a file needs to be worked on and yet have a back-up in case something happens to the file. In any case, the copy is made with the copy command (**cp**). To make a copy of a file named *good*, the following command will place a copy in a file named *savegood*:

```
cp good savegood
```

Two identical copies of the file *good* exist. If *savegood* previously contained something, it is overwritten.

To get the file *savegood* back to its original filename, *good*, the following commands are used:

```
mv savegood good
```

if *savegood* is not needed anymore or

```
cp savegood good
```

to retain a copy of *savegood*.

In summary, **mv** renames a file; **cp** makes a duplicate copy. Both commands overwrite the target file if one already exists unless write permission is denied by the mode of the file.



### 6.1.3 Combining Files

A familiar requirement is that of collecting two or more files into one big file, *bigfile*. This is needed, for example, when the author of a paper decides that several sections are to be combined. There are several ways to do this; the cleanest is a command called **cat** (not all commands have 2-letter names). The word **cat** is short for "concatenate", which is exactly what is desired. The command

```
cat file
```

prints the contents of the *file* on the terminal. The command

```
cat file1 file2
```

causes the contents of *file1* and *file2* to be printed on the terminal, in that order, but does not place them in *bigfile*.

There is a way to tell the system to put the same information in a file instead of printing on the terminal. The way to do it is to add to the command line the **>** character and the name of the file where the output is to go. The command

```
cat file1 file2 > bigfile
```

is used and the job is done. As with **cp** and **mv**, when something is put into *bigfile*, anything already there is destroyed. The ability to capture the output of a program can be used with any command that prints on a terminal. Several files can be combined, not just two.

```
cat file1 file2 file3 ... > bigfile
```

collects many individual files.

Sometimes a file needs to be appended to the end of another file. For example:

```
cat good good1 > temp  
mv temp good
```

is the most direct way. The following command:

```
cat good good1 > good
```

does not work because the **>** empties *good* before the **cat** program begins. The easiest way is to use a variant of **>**, called **>>**. In fact, **>>** is identical to **>** except that instead of clobbering the old file it adds something to the end. Thus the command

```
cat good1 >> good
```

adds *good1* to the end of *good*. If *good* does not exist, this makes a copy of *good1* called *good*.

### 6.1.4 Removing Files

If a file is not needed, it can be removed. The **rm** command

```
rm savegood
```

irrevocably deletes the file called *savegood* if the user had write permission.



## 6.2 Text Editor Functions

Manipulating pieces of files, individual lines, or groups of lines are handled with the text editor.

### 6.2.1 File Names

It is important to know the editor (**ed**) commands for reading and writing files. Equally useful is the edit command (**e**). Within **ed**, the command

```
e newfile
```

says "edit a new file called *newfile* without leaving the text editor". The **e** command discards whatever is being worked on and starts over on *newfile*. This is the same as if one had quit with the **q** command and reentered **ed** with a new file name except that if a pattern has been remembered, a command like **//** will still work.

When entering **ed** with the command

```
ed file
```

**ed** remembers the name of the file, and any subsequent **e**, **r**, or **w** commands that do not contain a file name will refer to this remembered file. Thus:

```
ed file1
----      (editing)
w          (writes back in file1)
e file2    (edit different file, without leaving ed)
----      (editing on file2)
w          (writes back on file2)
```

etc., does a series of edits on various files without leaving **ed** and without typing the name of any file more than once. By examining the sequence of commands in this example, it can be seen why many operating systems use **e** as a synonym for **ed**.

The current file name can be found at any time with the **f** command by typing **f** without a file name. Also, the name of a remembered file can be changed with **f**. A useful sequence is

```
ed precious
f junk
----      (editing)
```

This obtains a copy of the file **precious** and guarantees that a subsequent **w** command without a filename will write to *junk* and will not overwrite the original file.

### 6.2.2 Inserting One File Into Another

When a file is to be inserted into another, the **r** command can be used. For example, if the file *table* is to be inserted just after the reference to "Table 1", the following can be used:

```
/Table 1/
Table 1 shows that...      (response from ed)
.r table
```

The critical line is the last one. The **.r** command reads a file in after dot. An **r** command without any address adds lines to the end of the file, so it is equivalent to the **\$r** command.



### 6.2.3 Writing Out Part of a File

Another feature is writing to another file part of the document that is being edited. For example, it is possible to split into a separate file the table from the previous example, so it can be formatted or tested separately. If in the file being edited, there is

```

      --- text
    .TS
      --- lots of stuff
    .TE
      --- text

```

(which is the way a table is set up (as explained in Section III) to isolate the table in a separate file called *table*, first the start of the table (the .TS line) is found, and then the interesting part is written on file *table*:

```

/^\.TS/
.TS      (response from ed)
./^\.TE/w table

```

The same job can be accomplished with the single command

```

/^\.TS/;/^\.TE/w table

```

The point is that the *w* command can write out a group of lines instead of the whole file. A single line can be written by using one line number instead of two. For example, if a complicated line was just typed and it will be needed again, it should be saved and read in later rather than retyped:

```

a
      --- lots of stuff
      --- stuff to repeat
.
.w temp
a
      --- more stuff
.
.r temp
a
      --- more stuff

```

### 6.2.4 Moving Lines Around

Moving a paragraph from its present position in a paper to the end can be done several ways. For example, it is assumed that each paragraph in the paper begins with the formatting command ".PP". The brute force way (not necessarily bad) is to write the paragraph onto a temporary file, delete it from its current position, and then read in the temporary file at the end. If dot is at the ".PP" command that begins the paragraph, this is the sequence of commands:

```

./^\.PP/-w temp
./-d
$r temp

```

This states that from where dot is now until one line before the next ".PP" write onto file *temp*. The same lines are deleted and the file *temp* is read in at the end of the working file.



An easier way is to use the move command (**m**) that **ed** provides. This does the whole set of operations at one time without a temporary file. The **m** command is like many other **ed** commands in that it takes up to two line numbers in front to tell which lines are to be affected. It is also followed by a line number that tells where the lines are to go. Thus:

```
line1,line2m line3
```

says "move all the lines from **line1** through **line2** to after **line3**". Any of "**line1**", etc., can be patterns between slashes, dollar signs, or other ways to specify lines. If dot is at the first line of the paragraph, the command

```
./^\.PP/-m$
```

will also accomplish this task.

As another example of a frequent operation, the order of two adjacent lines can be reversed by moving the first one after the second. If dot is positioned at the first line, then

```
m+
```

does it. It says to move the line to after the dot. If dot is positioned on the second line:

```
m--
```

does the interchange.

The **m** command is more concise and direct than writing, deleting, and rereading. The main difficulty with the **m** command is that if patterns are used to specify both the line being moved and the target line, they must be specified properly or the wrong lines may be moved. The result of a botched **m** command can be a costly mistake. Doing the job a step at a time makes it easier to verify that each step accomplished what was wanted. It is also a good idea to issue a **w** command before doing anything complicated; then if an error is made, it is easy to back up.

### 6.2.5 Copying Lines

The **ed** program provides a transfer command (**t**) for making a copy of a group of one or more lines at any point. This is often easier than writing and reading. The **t** command is identical to the **m** command except instead of moving lines it duplicates them at the place referenced. Thus:

```
1,$t$
```

duplicates the entire contents that is being edited. A more common use for **t** is creating a series of lines that differ only slightly. For example:

```
a
    --- long line of stuff
```

```
t.          (make a copy)
s/x/y/      (change it a bit)
t.          (make third copy)
s/y/z/      (change it a bit)
```

### 6.2.6 Marks

The **ed** program provides for marking a line with a particular name so that the line can be referenced later by its name regardless of its line number. This can be useful for moving lines and for keeping track of them as they move. The mark command is **k**. The mark name must be a single lowercase letter. The command



kx

marks the current line with the name "x". If a line number precedes the k, that line is marked. The marked line can then be referred to with the address

'x

Marks are most useful for moving things around. The first line of the block to be moved is found and marked with ka. Then the last line is found and marked with kb. Dot is then positioned at the place where the lines are to go and the following command is performed:

'a,'bm.

**Note:** Only one line can have a particular mark name associated with it at any given time.

### 6.3 Temporary Escape

Sometimes it is convenient to temporarily escape from the text editor to do some UNIX operating system command without leaving the text editor. The escape command (!) provides a way to do this. If the command

!<any UNIX operating system command>

is entered, the current editing state is suspended; and the command asked for is executed. When the command finishes, ed will return a signal by printing another ! and editing can be resumed.

Any UNIX operating system command may be performed including another ed (this is quite common). In this case, another ! can be done.

## 7. Supporting Tools

There are several related tools and techniques which are relatively easy to learn after ed has been learned because they are based on ed. This section gives some cursory examples of these tools, more to indicate their existence than to provide a complete tutorial.

### 7.1 Global Printing From a Set of Files (grep)

Sometimes all occurrences of some word or pattern in a set of files need to be found in order to edit them or perhaps to verify their presence or absence. It may be possible to edit each file separately and look for the pattern of interest. If there are many files, this can be tedious; and if the files are really big, it may be impossible because of limits in ed.

The **grep** program was written to get around these limitations. Search patterns described in this section are often called "regular expressions", and "grep" stands for

g/re/p

This describes what **grep** does—it prints every line in a set of files that contains a particular pattern. Thus:

grep 'string' file1 file2 file3 ...

finds "string" wherever it occurs in any of the files *file1*, *file2*, etc. The **grep** program also indicates the file in which the line was found, so it can be edited later if needed.

The pattern represented by "string" can be any pattern that can be used in the text editor since **grep** and **ed** use the same mechanism for pattern searching. It is wisest to enclose the pattern in single quotes ('...') if



it contains any nonalphabetic characters since many such characters also mean something special to the UNIX operating system command interpreter (the "shell"). Without single quotes, the command interpreter will try to interpret them before **grep** has the opportunity.

There is also a way to find lines that do not contain a pattern:

```
grep -v 'string' file1 file2 ...
```

finds all lines that do not contain "string". The **-v** must occur in the position shown. Given **grep** and **grep -v**, it is possible to select all lines that contain some combination of patterns. For example, to obtain all lines that contain "x" but not "y":

```
grep x file... | grep -v y
```

The pipe notation (**|**) causes the output of the first command to be used as input to the second command.

## 7.2 Editing Scripts

If a fairly complicated set of editing operations is to be performed on an entire set of files, the easiest thing to do is to make a script file, i.e., a file that contains the operations to be performed and then apply this script to each file in turn. For example, if every instance of "This" needs to be changed to "THIS" and every instance of "That" needs to be changed to "THAT" in a large number of files, a file *script* is made with the following contents:

```
g/This/s//THIS/g
g/That/s//THAT/g
w
q
```

The following is done:

```
ed file1 <script
ed file2 <script
...
```

This causes **ed** to take its commands from the prepared script. The whole job has to be planned in advance.

By using the UNIX operating system command interpreter [**sh(1)**], a set of files can be cycled automatically with varying degrees of ease.



## STREAM EDITOR

### 1. Introduction

The stream editor (**sed**) is a noninteractive context editor that runs on the UNIX operating system. The **sed** software is designed to be especially useful in the following cases:

- When editing files too large for comfortable interactive editing
- When editing any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode
- When performing multiple global editing functions efficiently in one pass through the input file.

Because only a few lines of the input file reside in memory at one time and no temporary files are used, the effective size of a file that can be edited is limited only by the requirement that the input and output files fit simultaneously into available secondary storage.

Complicated editing scripts can be created separately and given to the **sed** program as a command file. For complex edits, this saves considerable typing and attendant errors. The **sed** program running from a command file is much more efficient than an interactive editor even if that editor can be driven by a prewritten script.

The principal loss of functions, if compared to an interactive editor, are lack of relative addressing (because of the line-at-a-time operation) and the lack of immediate verification that a command has done what was intended.

The **sed** program is a lineal descendant of the text editor, **ed**. Because of the differences between interactive and noninteractive operations, considerable changes have been made between **ed** and **sed**.

### 2. Overall Operation

The **sed** program by default copies the standard input to the standard output, perhaps performing one or more editing commands on each line before writing it to the output. This behavior may be modified by flags on the command line. (See Command Line Flags, paragraph 2.1.)

The general format of an editing command is

[address1,address2] function [arguments]

One or both addresses may be omitted. Any number of blanks or tabs may separate the addresses from the function. The function must be present. Arguments may be required or optional according to the function given. Tab characters and spaces at the beginning of lines are ignored.

#### 2.1 Command Line Flags

Three flags are recognized on the command line:

- n      tells the **sed** program not to copy all lines, but only those specified by **p** (print) functions or **s** (substitute) functions
- e      tells the **sed** program to take the next argument as an editing command
- f      tells the **sed** program to take the next argument as a file name; the file should contain editing commands—one to a line.



## 2.2 Order of Application of Editing Commands

Before any input file is opened, all editing commands are compiled into a form which will be moderately efficient during the execution phase (when the commands are actually applied to lines of the input file).

- Commands are compiled in the order encountered; generally, the order they will be attempted at execution time.
- Commands are applied one at a time; the input to each command is the output of all preceding commands.

The default linear order of application of editing commands can be changed by the `t` (test substitution) and `b` (branch) flow-of-control commands. When the order of application is changed by these commands, it remains true that the input line to any command is the output of any previously applied command.

## 2.3 Pattern Space

The range of pattern matches is called the pattern space. Ordinarily, the pattern space is one line of the input text, but more than one line can be read into the pattern space by using the next command (`N`).

## 2.4 Examples

Examples scattered throughout the following paragraphs use the following standard input text, except where noted:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

The command

```
2q
```

will copy the first two lines of the input and quit. The output will be

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
```

## 3. Selecting Lines for Editing

Input file lines that editing commands are to be applied to can be selected by addresses. Addresses may be either line numbers or context addresses.

The application of a group of commands can be controlled by one address (or address pair) by grouping commands with curly braces (`{ }`).

### 3.1 Line Number Addresses

A line number is a decimal integer. As each line is read from the input, a line number counter is incremented. A line number address matches (selects) the input line causing the internal counter to equal the address line number. The counter runs cumulatively through multiple input files. It is not reset when a new input file is opened. As a special case, the `$` character matches the last line of the last input file.



### 3.2 Context Addresses

A context address is a pattern (a regular expression) enclosed in slashes (/.../). Regular expressions recognized by the sed program are constructed as follows:

- An ordinary character is a regular expression and matches that character.
- A circumflex (^) at the beginning of a regular expression matches the null character at the beginning of a line.
- A dollar sign (\$) at the end of a regular expression matches the null character at the end of a line.
- The characters (\n) match an embedded newline character but not the newline character at the end of the pattern space.
- A period (.), sometimes called dot, matches any character except the terminal newline character of the pattern space.
- A regular expression followed by an asterisk (\*) matches any number (including 0) of adjacent occurrences of the regular expression it follows.
- A string of characters in square brackets ([ ]) matches any character in the string and no others. If, however, the first character of the string is a circumflex (^), the regular expression matches any character except the characters in the string and the terminal newline character of the pattern space. The circumflex is the only metacharacter recognized within the square brackets. If ] needs to be in the set of square brackets, it should be the first nonmetacharacter. For example:

|        |                    |
|--------|--------------------|
| [...]  | Includes ]         |
| [^...] | Does not include ] |

- A concatenation of regular expressions is a regular expression which matches the concatenation of strings matched by the components of the regular expression.
- A regular expression between the sequences \( and \) is identical in effect to the unadorned regular expression but has side effects which are described under the s command (substitute function) below.
- The expression \d means the same string of characters matched by an expression enclosed in \( and \) earlier in the same pattern. The d is a single digit; the string specified is that beginning with occurrence d of \( counting from the left. For example, the following expression matches a line beginning with two repeated occurrences of the same string:

```
^\(.*\)1
```

- The null regular expression standing alone (e.g., //) is equivalent to the last regular expression compiled.

To use one of the special characters (^ \$ . \* [ ] \ /) as a literal character (to match an occurrence of itself in the input), the special character is preceded by a backslash (\).

For a context address to match, the input requires that the whole pattern within the address match some portion of the pattern space.



### 3.3 Number of Addresses

Commands in the following paragraphs can have 0, 1, or 2 addresses. Under each command, the maximum number of allowed addresses is given. For a command to have more addresses than the maximum allowed is considered an error.

If a command has *no addresses*, it is applied to every line in the input.

If a command has *one address*, it is applied to all lines which match that address.

If a command has *two addresses*, it is applied to the first line which matches the first address and to all subsequent lines until (and including) the first subsequent line which matches the second address. An attempt is made on subsequent lines to again match the first address, and the process is repeated. Two addresses are separated by a comma. Some examples are:

/an/ matches lines 1, 3, and 4 in the sample text

/an.\*an/ matches line 1

/^an/ matches no lines

./ matches all lines

/\./ matches line 5

/r.\*an/ matches lines 1, 3, and 4 (number = 0)

/\ (an\).\*\1/ matches line 1.

## 4. Functions

Functions are named by a single alphabetic character. In the following function summaries, the maximum number of allowable addresses is enclosed in parentheses, followed by the single character function name. Possible arguments are enclosed in angle brackets (<>), and a description of each function is given. Angle brackets around arguments are not part of the argument and should not be typed in actual editing commands.

### 4.1 Whole Line Oriented Functions Summary

(2)d The **d** function deletes from the file (does not write to the output) those lines matched by its addresses. It also has the side effect that no further commands are attempted on the corpse of a deleted line. As soon as the **d** function is executed, a new line is read from the input, and the list of editing commands is restarted from the beginning on the new line.

(2)n The **n** function reads the next line from the input, replacing the current line, and the current line is written to the output. The list of editing commands is continued following the **n** command.

(1)a<text> The **a** function causes the argument <text> to be written to the output after the line matched by its address. The **a** command is inherently multiline; **a** must appear at the end of a line, and <text> may contain any number of lines. To preserve the one-command-to-a-line fiction, interior newline characters must be hidden by a backslash character (\) immediately preceding the newline character. The <text> is terminated by the first unhidden



newline character not immediately preceded by a backslash. Once an a function is successfully executed, <text> will be written to the output regardless of what later commands do to the line which triggered it. Even if that line is deleted, <text> will still be written to the output. The <text> is not scanned for address matches, and no editing commands are attempted on it. The a function does not cause a change in the line number counter.

(1)i\  
<text>

The i function behaves identically to the a function except that <text> is written to the output before the matched line. All other comments about the a function apply to the i function.

(2)c\  
<text>

The c function deletes lines selected by its addresses and replaces them with the lines in <text>. Like a and i, c must be followed by a newline character hidden by a backslash; interior newline characters in <text> must be hidden by backslashes. The c command may have two addresses, and therefore select a range of lines. If it does, all lines in the range are deleted, but only one copy of <text> is written to the output, not one copy per line deleted. As with a and i, <text> is not scanned for address matches, and no editing commands are attempted on it. It does not change the line number counter. After a line has been deleted by a c function, no further commands are attempted on the corpse. If text is appended after a line by a or r functions and the line is subsequently changed, the text inserted by the c function will be placed before the text of the a or r functions (the r function is described later).

For text put in the output by these functions, leading blanks and tabs will disappear as in sed commands. To get leading blanks and tabs into the output, the first desired blank or tab is preceded by a backslash. The backslash will not appear in the output. The list of editing commands for example:

```
n
a\
XXXX
d
```

applied to the standard input produces

```
In Xanadu did Kubla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
```

In this particular case, the same effect would be produced by either of the two following command lists:

```
n
i\
XXXX
d
```

or



```

n
c\
XXXX

```

## 4.2 Substitute Function

One important substitute function that changes parts of lines selected by a context search within the line is

```
(2)s<pattern><replacement><flags>
```

The **s** function replaces the part of a line selected by **<pattern>** with **<replacement>**. It can be read

```
Substitute for <pattern>, <replacement>
```

### 4.2.1 Pattern

The **<pattern>** argument contains a pattern exactly like the patterns in addresses. The only difference between **<pattern>** and a context address is that the context address must be delimited by slash (/) characters; **<pattern>** may be delimited by any character other than space or newline. By default, only the first string matched by **<pattern>** is replaced unless the **g** flag (below) is invoked.

### 4.2.2 Replacement

The **<replacement>** argument begins immediately after the second delimiting character of **<pattern>** and must be followed immediately by another instance of the delimiting character (thus there are exactly three instances of the delimiting character). The **<replacement>** is not a pattern, and the characters which are special in patterns do not have special meaning in **<replacement>**. Instead, other characters are special:

**\&** is replaced by the string matched by **<pattern>**.

**\d** is replaced by substring **d** (**d** is a single digit), matched by parts of **<pattern>**, and enclosed in **\(** and **\)**. If nested substrings occur in **<pattern>**, substring **d** is determined by counting opening delimiters **\(**. As in patterns, special characters may be made literal characters by preceding them with backslash (**\**).

### 4.2.3 Flags

The **<flags>** argument may contain the following:

- g** Substitute **<replacement>** for all nonoverlapping instances of **<pattern>** in the line. After a successful substitution, the scan for the next instance of **<pattern>** begins just after the end of the inserted characters. Characters put into the line from **<replacement>** are not rescanned.
- p** Print the line if a successful replacement was done. The **p** flag causes the line to be written to the output if and only if a substitution was actually made by the **s** function. If several **s** functions, each followed by a **p** flag, successfully substitute in the same input line, multiple copies of the line will be written to the output—one for each successful substitution.
- w <filename>** Write the line to a file if a successful replacement was done. The **w** flag causes lines which are actually substituted by the **s** function to be written to a file named by **<filename>**. If



<filename> exists before sed is run, it is overwritten; if not, it is created. A single space must separate **w** and <filename>. The possibilities of multiple, somewhat different copies of one input line being written are the same as for **p**. A maximum of ten different file names may be mentioned after **w** flags and **w** functions.

#### 4.2.4 Examples

The command

```
s/to/by/w changes
```

applied to the standard input produces on the output

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless by man
Down by a sunless sea.
```

and on the file *changes*

```
Through caverns measureless by man
Down by a sunless sea.
```

If the no-copy option is in effect, the command

```
s/[.,;?:]/*P&*/gp
```

produces

```
A stately pleasure dome decree*P:*
Where Alph*P,* the sacred river*P,* ran
Down to a sunless sea*P.*
```

To illustrate the effect of the **g** flag, the command

```
/X/s/an/AN/p
```

produces (assuming no-copy mode)

```
In XANadu did Kubla Khan
```

and the command

```
/X/s/an/AN/gp
```

produces

```
In XANadu did Kubla KhAN
```

#### 4.3 Input/Output Functions

- (2)**p** The *print* function writes addressed lines to the standard output file. They are written at the time the **p** function is encountered regardless of what succeeding editing commands may do to the lines.



(2)w <filename> The *write* function writes addressed lines to the file named by <filename>. If the file previously existed, it is overwritten; if not, it is created. The lines are written exactly as they exist when the write function is encountered for each line regardless of what subsequent editing commands may do to them. Exactly one space must separate the w and <filename>. A maximum of ten different files may be mentioned in write functions and w flags after s functions combined.

(1)r <filename> The *read* function reads the contents of <filename> and appends them after the line matched by the address. The file is read and appended regardless of what subsequent editing commands do to the line which matched its address. If r and a functions are executed on the same line, the text from a functions and r functions is written to the output in the order that the functions are executed. Exactly one space must separate the r and <filename>. If a file mentioned by an r function cannot be opened, it is considered a null file, not an error, and no diagnostic is given.

**Note:** Since there is a limit to the number of files that can be opened simultaneously, care should be taken that no more than ten files be mentioned in w functions or flags. That number is reduced by one if any r functions are present (only one read file is opened at a time).

If the file *notel* has the following contents

Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was the grandson and most eminent successor of Genghiz (Chingiz) Khan and founder of the Mongol dynasty in China.

then the command

/Kubla/r notel

produces

In Xanadu did Kubla Khan

Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was the grandson and most eminent successor of Genghiz (Chingiz) Khan and founder of the Mongol dynasty in China.

A stately pleasure dome decree:  
Where Alph, the sacred river, ran  
Through caverns measureless to man  
Down to a sunless sea.

#### 4.4 Multiple Input Line Functions

Three functions, all spelled with capital letters, deal with pattern spaces containing embedded newline characters. They are intended principally to provide pattern matches across lines in the input.

(2)N The next input line is appended to the current line in the pattern space. The two input lines are separated by an embedded newline character. Pattern matches may extend across embedded newline characters.



- (2)D Delete first part of the pattern space. Delete up to and including the first newline character in the current pattern space. If the pattern space becomes empty (the only newline character was the terminal newline character), read another line from the input. In any case, begin the list of editing commands again from the beginning.
- (2)P Print first part of the pattern space. Print up to and including the first newline character in the pattern space.

The P and D functions are equivalent to their lowercase counterparts if there are no embedded newline characters in the pattern space.

#### 4.5 Hold and Get Functions

Four functions save and retrieve part of the input for possible later use.

- (2)h Hold pattern space. The h function copies contents of the pattern space into a hold area destroying previous contents.
- (2)H Hold pattern space. The H function appends contents of the pattern space to contents of the hold area. Former and new contents are separated by a newline character.
- (2)g Get contents of hold area. The g function copies contents of the hold area into the pattern space destroying previous contents.
- (2)G Get contents of hold area. The G function appends contents of the hold area to contents of the pattern space. Former and new contents are separated by a newline character.
- (2)x Exchange. The exchange command interchanges contents of the pattern space and the hold area.

The following are examples:

```
1h
1s/did.*//
1x
G
s/\n/ :/
```

when applied to the standard input text, produce

```
In Xanadu did Kubla Khan :In Xanadu
A stately pleasure dome decree: :In Xanadu
Where Alph, the sacred river, ran :In Xanadu
Through caverns measureless to man :In Xanadu
Down to a sunless sea. :In Xanadu
```

#### 4.6 Flow of Control Functions

These functions do no editing on the input lines but control the application of functions to the lines selected by the address part.

- (2)! Don't  
The don't command causes the next command (written on the same line) to be applied to those input lines not selected by the address part.



- (2){      **Grouping**  
The grouping command causes the next set of commands to be applied (or not applied) as a block to the input lines selected by the addresses of the grouping command. The first of the commands under control of the grouping may appear on the same line as the { or on the next line. The group of commands is terminated by a matching } standing on a line by itself. Groups can be nested.
- (0):<label>      **Place a label**  
The label function marks a place in the list of editing commands which may be referred to by b and t functions. The <label> may be any sequence of eight or fewer characters. If two different colon functions have identical labels, a compile time diagnostic will be generated; and no execution attempted.
- (2)b<label>      **Branch to label**  
The branch function causes the sequence of editing commands being applied to the current input line to be restarted immediately after the place where a colon function with the same <label> was encountered. If no colon function with the same label can be found after all editing commands have been compiled, a compile time diagnostic is produced; and no execution is attempted. A b function with no <label> is a branch to the end of the list of editing commands. Whatever should be done with the current input line is done, and another input line is read. The list of editing commands is restarted from the beginning on the new line.
- (2)t<label>      **Test substitutions**  
The t function tests whether any successful substitutions have been made on the current input line; if so, it branches to <label>; if not, it does nothing. The flag which indicates that a successful substitution has been executed is reset by reading a new input line and executing a t function.

#### 4.7 Miscellaneous Functions

- (1)=      The = function writes to standard output the line number of the line matched by its address.
- (1)q      The q function causes the current line to be written to the output (if it should be), any appended or read text to be written, and execution to be terminated.



## MISCELLANEOUS FACILITIES

Several miscellaneous facilities exist (via UNIX operating system commands) to aid in the development of documentation. These facilities are easy to access and are very effective. Their use is beneficial in documentation development. Some available miscellaneous facilities are described briefly in the following list. The User's Manual—UNIX Operating System has a more detailed description.

- bdiff** The **bdiff** facility is used in a manner analogous to **diff** to find which lines must be changed in two files to bring them into agreement. Its purpose is to allow processing of files which are too large for **diff**.
- cat** The **cat** facility reads each file in sequence and writes it on the standard output. Thus:
- cat file**
- prints the file named *file*, and
- cat file1 file2 > file3**
- concatenates *file1* and *file2* and places the result in *file3*.
- cmp** The **cmp** facility compares two files. Under default options, **cmp** makes no comment if the files are the same; if they differ, it announces the byte and line number at which the difference occurred.
- comm** The **comm** facility selects or rejects lines common to two sorted files. It reads *file1* and *file2* and produces a 3-column output as follows: lines only in *file1*, lines only in *file2*, and lines in both files.
- diff** The **diff** facility is a differential file comparator. It tells what lines must be changed in two files to bring them into agreement.
- diff3** The **diff3** facility is a 3-way differential file (files up to 64K) comparator. It compares three versions of a file and publishes disagreeing ranges of text flagged with special codes.
- diffmk** The **diffmk** facility marks the differences between files. It compares two versions of a file and creates a third file that includes "change mark" commands for the **nroff** or **troff** formatter.
- grep** Commands of the **grep** facility search the input files for lines matching a pattern. Normally, each line found is copied to the standard output. The **grep** patterns are limited regular expressions in the style of **ed**. The **egrep** patterns are full regular expressions. The **fgrep** patterns are fixed strings.
- pr** The **pr** facility prints the named files on the standard output. If file is — or if no files are specified, the standard input is assumed.
- sdiff** The **sdiff** facility uses the output of **diff(1)** to produce a side-by-side listing of two files indicating those lines that are different. Each line of the two files are printed with a blank gutter between them if the lines are identical, a > in the gutter if the line exists only in *file1*, a < in the gutter if the line exists only in *file2*, and a ! for lines that are different.
- sort** The **sort** facility sorts lines of all the named files together and writes the results on the standard output.



- spell** The **spell** facility collects words from the named files and looks them up in a spelling list. Words that do not occur in the spelling list nor can be derived from them are printed on the standard output. The **spellin** and **spellout** are two additional subroutines of **spell**.
- split** The **split** facility splits a file into pieces.
- typo** The **typo** facility searches through a document for unusual words, typographical errors, and hapax legomena and prints them on the standard output.
- uniq** The **uniq** facility reports repeated lines in a file. It reads the input file comparing adjacent lines. In the normal case, the second and succeeding copies of repeated lines are removed; the remainder is written on the output file.



### III. FORMATTING FACILITIES

#### NROFF AND TROFF USER'S MANUAL

##### 1. Introduction

Text processors, **nroff** and **troff**, under the UNIX operating system format text for typewriter-like terminals and for a phototypesetter, respectively. Both **nroff** and **troff** processors accept lines of text interspersed with lines of format control information. They format the text into a printable, paginated document having a user-designed style. The **nroff** and **troff** formatters offer unusual freedom in document styling including:

- Arbitrary style headers and footers
- Arbitrary style footnotes
- Multiple automatic sequence numbering for paragraphs and sections
- Multiple column output
- Dynamic font and point-size control
- Arbitrary horizontal and vertical local motions at any point
- Overstriking, bracket construction, and line drawing functions.

Since **nroff** and **troff** formatters are reasonably compatible, it is usually possible to prepare input acceptable to both. Conditional input is provided that enables the user to embed input expressly destined for either program. The **nroff** formatter can prepare output directly for a variety of terminal types and is capable of utilizing the full resolution of each terminal.

The **troff** processor is a text-formatting program for driving a phototypesetter on the UNIX operating system. It is capable of producing high quality text. The phototypesetter normally runs with four fonts containing Roman, italic, and bold letters; a full Greek alphabet; a substantial number of special characters; and mathematical symbols. Characters can be printed in a range of sizes and placed anywhere on the page.

Full user control over fonts, sizes, and character positions, as well as the usual features of a formatter (right-margin justification, automatic hyphenation, page titling and numbering, etc.) are provided by the **troff** processor. It also provides macros, arithmetic variables and operations, and conditional testing for complicated formatting tasks.

##### 2. Usage

The general form of invoking an **nroff** or **troff** formatter at the UNIX operating system command level is

```
nroff options files  
or  
troff options files
```

where *options* represents any of a number of option arguments and *files* represents the list of files containing the document to be formatted. An argument consisting of a single minus sign (-) is taken to be a file name corresponding to the standard input. Input is taken from the standard input if no file names are given. Options may appear in any order so long as they appear before the files.



nroff and troff

| OPTION         | EFFECT                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>-olist</b>  | Prints only pages whose page numbers appear in <i>list</i> , which consists of comma-separated numbers and number ranges. <ul style="list-style-type: none"> <li>• A number range has the form <i>N-M</i> and means pages <i>N</i> through <i>M</i></li> <li>• An initial <i>-N</i> means from the beginning to page <i>N</i></li> <li>• A final <i>N-</i> means from page <i>N</i> to the end.</li> </ul>                                                                                                                                                                                                     |
| <b>-nN</b>     | Number the first generated page <i>N</i> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>-sN</b>     | Stop every <i>N</i> page (and cause the bell control character to be output to the terminal). The <b>nroff</b> formatter will halt after every <i>N</i> pages (default <i>N</i> =1) to allow paper loading or changing and will resume upon receipt of a new line. The <b>troff</b> formatter will stop the phototypesetter every <i>N</i> pages, produce a trailer to allow changing cassettes, and resume after the phototypesetter START button is pressed.                                                                                                                                                 |
| <b>-m name</b> | Prepend the macro file<br><div style="text-align: center;"><b>/usr/lib/tmac/tmac.name</b></div> to the input files. Multiple <b>-m</b> macro package requests on a command line are accepted and are processed in sequence.                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>-c name</b> | Prepend the macro files<br><div style="text-align: center;"><b>/usr/lib/macros/cmp.[nt].[dt].name</b></div> and<br><div style="text-align: center;"><b>/usr/lib/macros/ucmp.[nt].name</b></div> to the input files. Multiple <b>-c</b> macro package requests on a command line are accepted. The compacted version of macro package <i>name</i> should be used if it exists. If not, the <b>nroff/troff</b> formatter will try the equivalent <b>-m name</b> option instead. This option should be used instead of <b>-m</b> because it makes the <b>nroff/troff</b> formatters execute significantly faster. |
| <b>-raN</b>    | Set register <i>a</i> (one character) to <i>N</i> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>-i</b>      | Read standard input after the input files are exhausted.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>-q</b>      | Invoke the simultaneous input/output mode of the <b>rd</b> request.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>-z</b>      | Suppress formatted output. Only message output will occur (from <b>tm</b> requests and diagnostics).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <b>-k name</b> | Produce a compacted macro package from this invocation of the <b>nroff/troff</b> formatter. This option has no effect if no <b>.co</b> request is used in the <b>nroff/troff</b> formatter input. Otherwise, the compacted output is produced in files <i>d.name</i> and <i>t.name</i> .                                                                                                                                                                                                                                                                                                                       |



nroff Only

| OPTION        | EFFECT                                                                                                                                                                                                                                                                                                          |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>-Tname</b> | Specify the name of the output terminal type. Currently defined names are: <b>37</b> (default) for the TELETYPE Model 37, <b>tn300</b> for the GE TerminiNet 300 (or any terminal without half-line capabilities), <b>300</b> for the DASI 300, <b>300s</b> for the DASI 300s, and <b>450</b> for the DASI 450. |
| <b>-e</b>     | Produce equally spaced words in adjusted lines using full terminal resolution.                                                                                                                                                                                                                                  |
| <b>-h</b>     | Use output tabs during horizontal spacing to speed output and to reduce output byte count. Device tab settings are assumed to be every eight nominal character widths. The default settings of logical input tabs are also every eight nominal character widths.                                                |
| <b>-un</b>    | Set the emboldening factor (number of character overstrikes) in the <b>nroff</b> formatter for the third font position (bold) to be <i>n</i> (zero if <i>n</i> is missing).                                                                                                                                     |

troff Only

| OPTION     | EFFECT                                                                                                                                                                                         |
|------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>-t</b>  | Direct output to the standard output instead of the phototypesetter.                                                                                                                           |
| <b>-f</b>  | Refrain from feeding paper and stopping phototypesetter at the end of the run.                                                                                                                 |
| <b>-w</b>  | Wait until phototypesetter is available if busy.                                                                                                                                               |
| <b>-b</b>  | Report whether phototypesetter is busy or available. No text processing is done.                                                                                                               |
| <b>-a</b>  | Send a printable approximation in American Standard Code for Information Interchange (ASCII) character set of the results to the standard output. This approximates a display of the document. |
| <b>-pN</b> | Print all characters in point size <i>N</i> while retaining all prescribed spacings and motions to reduce phototypesetter elapsed time.                                                        |
| <b>-g</b>  | Prepare output for the Murray Hill Computation Center phototypesetter and direct it to the standard output.                                                                                    |

Each option is invoked as a separate argument. For example:

```
rnoff -o4,8-10 -T300s -mabc file1 file2
```

requests formatting of pages 4, 8, 9, and 10 of a document contained in the files named *file1* and *file2*, specifies the output terminal as a DASI 300s, and invokes the macro package *abc*.

Various preprocessors and postprocessors are available for use with the **nroff** and **troff** formatters:

- The equation preprocessors are **neqn** and **eqn** (for **nroff** and **troff** formatters, respectively).
- The table-construction preprocessor is **tbl**.
- A reverse-line postprocessor for multiple-column **nroff** formatter output on terminals without reverse-line ability is **col**. The TELETYPE Model 37 escape sequences that the **nroff** formatter produces by default are expected by **col**.



- The TELETYPE Model 37-simulator postprocessor for printing **nroff** formatter output on a Tektronix 4014 is **4014**.
- The phototypesetter-simulator postprocessor for the **troff** formatter that produces an approximation of phototypesetter output on a Tektronix 4014 is **tc**. For example, in

```
tbl files | eqn | troff -t [options] | tc
```

the first **|** indicates the piping of **tbl** output to **eqn** input; the second **|** indicates the piping of **eqn** output to the **troff** formatter input; and the third **|** indicates the piping of the **troff** formatter output to file **tc**.



### 3. NROFF/TROFF Reference Manual

#### 3.1 General Explanation

##### 3.1.1 Form of Input

Input consists of *text lines*, which are destined to be printed, interspersed with *control lines*, which set parameters or otherwise control subsequent processing. Control lines begin with a control character, normally a period or an acute accent, followed by a 1- or 2-character name that specifies a basic request or the substitution of a user-defined macro in place of the control line. The acute accent control character suppresses the break function (the forced output of a partially filled line) caused by certain requests. Control characters may be separated from request/macro names by white space (spaces and/or tabs) for esthetic reasons. Names must be followed by either a space or a newline character. Control lines with unrecognized request/macro names are ignored. Table 3.A is a cross reference of request names to the table in this section where an explanation of the request is displayed.

Various special functions may be introduced anywhere in the input by means of an escape character (\). For example, the function `\nR` causes the interpolation of the contents of the number register *R* in place of the function. Number register *R* is either an *x* for a single letter register name or (*xx* for a 2-character register name. Table 3.B itemizes escape sequences for characters, indicators, and functions.

##### 3.1.2 Formatter and Device Resolution

The **troff** processor internally uses 432 units/inch, corresponding to the Wang Laboratories phototypesetter which has a horizontal resolution of 1/432 inch and a vertical resolution of 1/144 inch. It rounds horizontal/vertical numerical parameter input to the actual horizontal/vertical resolution of the typesetter.

The **nroff** processor internally uses 240 units/inch, corresponding to the least common multiple of the horizontal and vertical resolutions of various typewriter-like output devices. It rounds numerical input to the actual resolution of the output device indicated by the `-T` option (default Model 37 Teletype).

##### 3.1.3 Numerical Parameter Input

Both **nroff** and **troff** formatters accept numerical input with the appended scale indicators shown in the following table, where *S* is the current type size in points, *V* is the current vertical line spacing in basic units, and *C* is a nominal character width in basic units.

| SCALE<br>INDICATOR | MEANING                      | NUMBER OF BASIC UNITS |                              |
|--------------------|------------------------------|-----------------------|------------------------------|
|                    |                              | TROFF                 | NROFF                        |
| i                  | Inch                         | 432                   | 240                          |
| c                  | Centimeter                   | 432x50/127            | 240x50/127                   |
| P                  | Pica = 1/6 inch              | 72                    | 240/6                        |
| m                  | em = <i>S</i> points         | 6x <i>S</i>           | <i>C</i>                     |
| n                  | en = em/2                    | 3x <i>S</i>           | <i>C</i> , same as <i>em</i> |
| p                  | Point = 1/72 inch            | 6                     | 240/72                       |
| u                  | Basic unit                   | 1                     | 1                            |
| v                  | Vertical line space          | <i>V</i>              | <i>V</i>                     |
| none               | Default (see following text) |                       |                              |



In **nroff** processors, both **em** and **en** are taken to be equal to *C*, which is output-device-dependent; common values are 1/10 and 1/12 inch. Actual character widths in the **nroff** formatter need not be all the same. Constructed characters (such as **->**) are often extra wide. Default scaling is:

- **em** for horizontally oriented requests (**.ll**, **.in**, **.ti**, **.ta**, **.lt**, **.po**, **.mc**) and functions (**\h**, **\l**).
- **V** for vertically oriented requests (**.pl**, **.wh**, **.ch**, **.dt**, **.sp**, **.sv**, **.ne**, **.rt**) and functions (**\v**, **\x**, **\L**)
- **p** for **.vs** request
- **u** for **.nr**, **.if**, and **.ie** requests.

All other requests ignore scale indicators. When a number register containing an already appropriately scaled number is interpolated to provide numerical input, the unit scale indicator (**u**) may need to be appended to prevent an additional inappropriate default scaling. The number, *N*, may be specified in decimal-fraction form but the parameter finally stored is rounded to an integer number of basic units.

The absolute position indicator (**l**) may be prepended to a number *N* to generate the distance to the vertical or horizontal place *N*.

- For vertically oriented requests and functions, **lN** becomes the distance in basic units from the current vertical place on the page or in a diversion (paragraph 3.7) to the vertical place *N*.
- For all other requests and functions, **lN** becomes the distance from the current horizontal place on the input line to the horizontal place *N*.

For example

**.sp 13.2c**

will space in the required direction to 3.2 centimeters from the top of the page.

### 3.1.4 Numerical Expressions

Wherever numerical input is expected, an expression involving parentheses, the arithmetic operators **+**, **-**, **/**, **\***, **%** (mod), and the logical operators **<**, **>**, **<=**, **>=**, **=** (or **=**), **&** (and), **:** (or) may be used. Except where controlled by parentheses, evaluation of expressions is left to right; there is no operator precedence. In the case of certain requests, an initial **+** or **-** is stripped and interpreted as an increment or decrement indicator. In the presence of default scaling, the desired scale indicator must be attached to every number in an expression for which the desired and default scaling differ. For example, if the number register **x** contains 2 and the current point size is 10, then:

**.ll (4.25i+\nxP+3)/2u**

will set the line length to ½ the sum of 4.25 inches + 2 picas + 3 ems (30 points since the point size is 10).

### 3.1.5 Notation

Numerical parameters are indicated in this manual in two ways. A **±N** means that the argument may take the forms **N**, **+N**, or **-N** and that the corresponding effect is to set the affected parameter to *N*, to increment it by *N*, or to decrement it by *N*, respectively. Plain *N* means that an initial algebraic sign is not an increment indicator but merely the sign of *N*. Generally, unreasonable numerical input is either ignored or truncated to a reasonable value. For example, most requests expect to set parameters to non-negative values; exceptions are



.sp, .wh, .ch, .nr, and .if. The requests .ps, .ft, .po, .vs, .ls, .ll, .in, and .lt restore the previous parameter value in the absence of an argument.

Single character arguments are indicated by single lowercase letters and 1- or 2-character arguments are indicated by a pair of lowercase letters. Character string arguments are indicated by multicharacter mnemonics.

### 3.2 Font and Character Size Control

#### 3.2.1 Fonts

Default mounted fonts are Times Roman (R), Times Italic (I), Times Bold (B), and Special Mathematical (S) on physical typesetter positions 1, 2, 3, and 4, respectively. These font styles are shown in Fig. 3.1. The current font, initially Times Roman, may be changed (among the mounted fonts) by use of the .ft request or by imbedding at any desired point either \fx, \f(xx, or \fN where x and xx are the name of a mounted font and N is a numerical font position. It is not necessary to change to the Special Font; characters on that font are automatically handled. A request for a named but not mounted font is ignored.

The troff processor can be informed that any particular font is mounted by use of the .fp request. The list of known fonts is installation dependent. In the subsequent discussion of font-related requests, F represents either a 1- or 2-character font name or the numerical font position, 1 through 4. The current font is available as numerical position in the read-only number register .f.

Font control is understood by the nroff formatter which normally underlines italic characters. Table 3.C is a summary and explanation of font control requests.

#### 3.2.2 Character Set

The troff character set consists of the so-called Commercial II character set plus a Special Mathematical font character set each having 102 characters. All ASCII characters are included with some on the Special Mathematical font. The ASCII characters are input as themselves (with three exceptions); and non-ASCII characters are input in the form \ (xx, where xx is a 2-character name given in Table 3.D. The three ASCII character exceptions are mapped as follows:

| ASCII INPUT |              | PRINTED BY TROFF |             |
|-------------|--------------|------------------|-------------|
| CHARACTER   | NAME         | CHARACTER        | NAME        |
| '           | acute accent | '                | close quote |
| `           | grave accent | `                | open quote  |
| -           | minus        | -                | hyphen      |

The characters ' , and - may be input by \', \`, and \-, respectively, or by their names. The ASCII characters @, #, " , ' , < , > , \ , { , } , ~ , ^ , and \_ exist on the Special Mathematical font and are printed as a one em space if that font is not mounted.

The nroff processor understands the entire troff character set but can print only:

- ASCII characters
- Additional characters as may be available on the output device



- Such characters as may be able to be constructed by overstriking or other combinations
- Those characters that can reasonably be mapped into other printable characters.

The exact behavior is determined by a driving table prepared for each device. The characters `'`, ```, and `_` print as themselves.

### 3.2.3 Character Size

Character point sizes available are 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, and 36. This is a range of 1/12 inch to 1/2 inch. The `.ps` request is used to change or restore the point size. Alternatively, the point size may be changed between any two characters by imbedding a `\sN` at the desired point to set the size to  $N$  or a `\s±N` ( $1 ≤ N ≤ 9$ ) to increment/decrement the size by  $N$ ; `\s0` restores the previous size. Requested point size values that are between two valid sizes yield the larger of the two. The current size is available in the `.s` number register. The `nroff` formatter ignores type size control. Table 3.E is a summary and explanation of character size requests.

### 3.3 Page Control

Top and bottom margins are not automatically provided. They may be defined by two macros which set traps at vertical positions 0 (top) and  $-N$  ( $N$  from the bottom). A pseudo-page transition onto the first page occurs either when the first break occurs or when the first nondiverted text processing occurs. Arrangements for a trap to occur at the top of the first page must be completed before this transition. A summary and explanation of page control requests is shown in Table 3.F. References to the current diversion mean that the mechanism being described works during both ordinary and diverted output (the former is considered as the top diversion level).

Usable page width on the phototypesetter is about 7.54 inches. The left margin begins about 1/27 inch from the edge of the 8-inch wide, continuous roll paper. Physical limitations on the `nroff` processor output are output-device-dependent.

### 3.4 Text Filling, Adjusting, and Centering

#### 3.4.1 Filling and Adjusting

Normally, words are collected from input text lines and assembled into an output text line until some word does not fit. An attempt may be made to hyphenate the word in an effort to assemble a part of it into the output line. The spaces between the words on the output line are increased to spread out the line to the current line length minus any current indent. A *word* is any string of characters delimited by the *space* character or the beginning/end of the input line. Any adjacent pair of words that must be kept together (neither split across output lines nor spread apart in the adjustment process) can be tied together by separating them with the *unpaddable space* backslash-space character (`\`). The adjusted word spacings are uniform in the `troff` formatter, and the minimum interword spacing can be controlled with the `.ss` request. In the `nroff` formatter, they are normally nonuniform because of quantization to character-size spaces; however, the command line option `-e` causes uniform spacing with full output device resolution. Filling, adjustment, and hyphenation can all be prevented or controlled. The text length on the last line output is available in the `.n` number register, and text base-line position on the page for this line is in the `.nl` number register. The text base-line high-water mark (lowest place) on the current page is in the `.h` register.

An input text line ending with `.`, `?`, or `!` is taken to be the end of a sentence, and an additional space character is automatically provided during filling. Multiple interword space characters found in the input are retained, except for trailing spaces; initial spaces also cause a break.

When filling is in effect, a `\p` escape sequence may be imbedded in or attached to a word to cause a break at the end of the word and have the resulting output line spread out to fill the current line length.



A text input line that happens to begin with a control character can be made not to look like a control line by prefacing it with the nonprinting, zero-width filler character (`\&`). Another way is to specify output translation of some convenient character into the control character using the `.tr` request.

### 3.4.2 Interrupted Text

Copying of an input line in no-fill mode can be interrupted by terminating the partial line with a `\c` escape sequence. The next encountered input text line will be considered to be a continuation of the same line of input text. Similarly, a word within filled text may be interrupted by terminating the word (and line) with `\c`; the next encountered text will be taken as a continuation of the interrupted word. If the intervening control lines cause a break, any partial line will be forced out along with any partial word.

Table 3.G is a summary and explanation of filling, adjusting, and centering requests.

## 3.5 Vertical Spacing

### 3.5.1 Base-line Spacing

Vertical spacing size ( $V$ ) between base lines of successive output lines can be set using the `.vs` request with a resolution of  $1/144$  inch =  $\frac{1}{2}$  point in the `troff` formatter and to the output device resolution in the `nroff` formatter. Spacing size must be large enough to accommodate character sizes on affected output lines. For the common type sizes (9 through 12 points), usual typesetting practice is to set  $V$  to two points greater than the point size; `troff` default is 10-point type on a 12-point spacing. The current  $V$  is available in the `.v` register. Multiple- $V$  line separation (e.g., double spacing) may be obtained with a `.ls` request.

### 3.5.2 Extra Line Space

If a word contains a vertically tall construct requiring the output line containing it to have extra vertical space before and/or after it, the *extra line space* function `\x'N'` can be imbedded in or attached to that word. In this and other functions having a pair of delimiters around their parameter, the delimiter choice is arbitrary except that it can not look like the continuation of a number expression for  $N$ .

- If  $N$  is negative, the output line containing the word will be preceded by  $N$  extra vertical spaces.
- If  $N$  is positive, the output line containing the word will be followed by  $N$  extra vertical spaces.
- If successive requests for extra space apply to the same line, the maximum values are used.

The most recently utilized post-line extra line space is available in the `.a` register.

### 3.5.3 Blocks of Vertical Space

A block of vertical space is ordinarily requested using `.sp`, which honors the no-space mode and which does not space past a trap. A contiguous block of vertical space may be reserved using the `.sv` request.

Table 3.H is a summary and explanation of vertical spacing requests.

## 3.6 Line Length and Indenting

The maximum line length for fill mode may be set with a `.ll` request. The indent may be set with a `.in` request; an indent applicable to only the next output line may be set with the `.ti` request. The line length includes indent space but not page offset space. The line length minus the indent is the basis for centering with the `.ce` request. If a partially collected line exists, the effect of `.ll`, `.in`, or `.ti` is delayed until after that line is output.



In fill mode, the length of text on an output line is less than or equal to the line length minus the indent. The current line length and indent are available in registers `.l` and `.i`, respectively. The length of 3-part titles produced by `.tl` is independently set by `.lt`. Table 3.I is a summary and explanation of line length and indenting requests.

### 3.7 Macros, Strings, Diversions, and Position Traps

#### 3.7.1 Macros and Strings

A macro is a named set of arbitrary lines that may be invoked by name or with a trap. A string is a named string of characters, not including a newline character, that may be interpolated by name at any point. Request, macro, and string names share the same name list. Macro and string names may be 1- or 2-characters long and may usurp previously defined request, macro, or string names. Any of these entities may be renamed with `.rn` or removed with `.rm`.

- Macros are created by `.de` and `.di` and appended by `.am` and `.da` (`.di` and `.da` cause normal output to be stored in a macro)
- Strings are created by `.ds` and appended by `.as`.

A macro is invoked in the same way as a request; a control line beginning `.xx` will interpolate the contents of macro `xx`. The remainder of the line may contain up to nine arguments. The strings `x` and `xx` are interpolated at any desired point with `\*x` and `\*(xx`, respectively. String references and macro invocations may be nested.

#### 3.7.2 Copy Mode Input Interpretation

During the definition and extension of strings and macros (not by diversion), the input is read in copy mode. The input is copied without interpretation except that:

- Contents of number registers indicated by `\n` are interpolated.
- Strings indicated by `\*` are interpolated.
- Arguments indicated by `\$` are interpolated.
- Concealed newline characters indicated by `\<newline>` are eliminated.
- Comments indicated by `\ "` are eliminated.
- `\t` and `\a` are interpreted as ASCII horizontal tab and start of heading (SOH), respectively (Part 9).
- `\\` is interpreted as `"\"`.
- `\.` is interpreted as `"."`.

These interpretations can be suppressed by prepending a `\`. For example, since `\\` maps into a `\`, `\\n` will copy as `\n` which will be interpreted as a number register indicator when the macro or string is reread.

#### 3.7.3 Arguments

When a macro is invoked by name, the remainder of the line may contain up to nine arguments. The argument separator is the space character, and arguments may be surrounded by double-quotes to permit imbedded space characters. Pairs of double-quotes may be imbedded in double-quoted arguments to represent a single



double-quote. If the desired arguments will not fit on a line, a concealed newline character may be used to continue on the next line.

When a macro is invoked, the input level is pushed down and any arguments available at the previous level become unavailable until the macro is completely read and the previous level is restored. A macro's own arguments can be interpolated at any point within the macro with  $\backslash \$N$ , which interpolates the  $N$ th argument ( $1 \leq N \leq 9$ ). If an invoked argument does not exist, a null string results. For example, the macro *xx* may be defined by

```
.de xx      \ " begin definition
Today is \\\$1 the \\\$2.
..         \ " end definition
```

and called by

```
.xx Monday 14th
```

to produce the text

```
Today is Monday the 14th.
```

The  $\backslash \$$  was concealed in the definition with a prepended  $\backslash$ . The number of currently available arguments is in the  $\$.$  register.

No arguments are available at the top (nonmacro) level in this implementation. Because string referencing is implemented as an input-level pushdown, no arguments are available from within a string. No arguments are available within a trap-invoked macro.

Arguments are copied in copy mode onto a stack where they are available for reference. The mechanism does not allow an argument to contain a direct reference to a long string (interpolated at copy time), and it is advisable to conceal string references (with an extra  $\backslash$ ) to delay interpolation until argument reference time.

### 3.7.4 Diversions

Processed output may be diverted into a macro for purposes such as footnote processing or determining the horizontal and vertical size of some text for conditional changing of pages or columns. A single diversion trap may be set at a specified vertical position. The number registers *.dn* and *.dl*, respectively, contain the vertical and horizontal size of the most recently ended diversion. Processed text that is diverted into a macro retains the vertical size of each of its lines when reread in no-fill mode regardless of the current *V*. Constant-spaced (*.cs*) or emboldened (*.bd*) text that is diverted can be reread correctly only if these modes are again or still in effect at reread time. One way to do this is to imbed in the diversion the appropriate *.cs* or *.bd* request with the transparent mechanism described in paragraph 3.10.6.

Diversions may be nested and certain parameters and registers are associated with the current diversion level (the top non-diversion level may be thought of as diversion level 0). These parameters and registers are:

- diversion trap and associated macro
- no-space mode
- internally saved marked place (see *.mk* and *.rt*)
- current vertical place (*.d* register)
- current high-water text base line (*.h* register)



- current diversion name (.z register).

### 3.7.5 Traps

Three types of trap mechanisms are available:

- page trap
- diversion trap
- input-line-count trap.

Macro-invocation traps may be planted using `.wh` requests at any page position including the top. This trap position may be changed using `.ch`. Trap positions at or below the bottom of the page have no effect unless or until moved to within the page or rendered effective by an increase in page length. Two traps may be planted at the same position only by first planting them at different positions and then moving one of the traps; the first planted trap will conceal the second unless and until the first one is moved. If the first planted trap is moved back, it again conceals the second trap. The macro associated with a page trap is automatically invoked when a line of text is output whose vertical size reaches or sweeps past the trap position. Reaching the bottom of a page springs the top-of-page trap, if any, provided there is a next page. The distance to the next trap position is available in the `.t` register; if there are no traps between the current position and the bottom of the page, the distance returned is the distance to the page bottom.

Macro-invocation traps, effective in the current diversion, may be planted using `.dt` requests. The `.t` register works in a diversion. If there is no subsequent trap, a large distance is returned.

Table 3.J is a summary and explanation of macros, strings, diversion, and position traps requests.

### 3.8 Number Registers

A variety of predefined number registers (Table 3.K) are available to the user. In addition, the user may define his own named registers. Register names are 1- or 2-characters long and do not conflict with request, macro, or string names. Except for certain predefined read-only number registers (Table 3.L), a number register can be read, written, automatically incremented or decremented, and interpolated into the input in a variety of formats. One common use of user-defined registers is to automatically number sections, paragraphs, lines, etc. A number register may be used any time numerical input is expected or desired and may be used in numerical expressions.

Number registers are created and modified using the `.nr` request, which specifies name, numerical value, and automatic increment size. Registers are also modified if accessed with an automatic incrementing sequence. If the registers `x` and `xx` both contain  $N$  and have the automatic increment size  $M$ , the following access sequences have the effect shown as follows:

| SEQUENCE            | EFFECT ON REGISTER      | VALUE INTERPOLATED |
|---------------------|-------------------------|--------------------|
| <code>nx</code>     | none                    | $N$                |
| <code>n(xx)</code>  | none                    | $N$                |
| <code>n+x</code>    | $x$ incremented by $M$  | $N+M$              |
| <code>n-x</code>    | $x$ decremented by $M$  | $N-M$              |
| <code>n+(xx)</code> | $xx$ incremented by $M$ | $N+M$              |
| <code>n-(xx)</code> | $xx$ decremented by $M$ | $N-M$              |



According to the format specified by the .af request, a number register is converted (when interpolated) to:

- decimal (default)
- decimal with leading zeros
- lowercase Roman
- uppercase Roman
- lowercase sequential alphabetic
- uppercase sequential alphabetic.

Table 3.M is a summary and explanation of number registers requests.

### 3.9 Tabs, Leaders, and Fields

#### 3.9.1 Tabs and Leaders

The ASCII horizontal tab character and the ASCII SOH character (the leader) can both be used to generate either horizontal motion or a string of repeated characters. The length of the generated entity is governed by internal tab stops specified with a .ta request. The default difference is that tabs generate motion and leaders generate a string of periods; .tc and .lc offer the choice of repeated character or motion. There are three types of internal tab stops: left justified, right justified, and centered. In the following table:

- *next-string* consists of the input characters following the tab (or leader) up to the next tab (or leader) or end of line
- *D* is the distance from the current position on the input line (where a tab or leader was found) to the next tab stop
- *W* is the width of *next-string*.

| TAB TYPE | LENGTH OF MOTION OR REPEATED CHARACTERS | LOCATION OF <i>next-string</i> |
|----------|-----------------------------------------|--------------------------------|
| Left     | $D$                                     | Following $D$                  |
| Right    | $D - W$                                 | Right justified within $D$     |
| Centered | $D - W/2$                               | Centered on right end of $D$   |

The length of generated motion is allowed to be negative but that of a repeated character string cannot be. Repeated character strings contain an integer number of characters, and any residual distance is prepended as motion. Tabs (or leaders) found after the last tab stop are ignored, but they may be used as *next-string* terminators.

Tabs and leaders are not interpreted in copy mode. The \t and \a always generate a noninterpreted tab and leader, respectively, and are equivalent to actual tabs and leaders in copy mode.



### 3.9.2 Fields

A field is contained between a pair of field delimiter characters. It consists of substrings separated by padding indicator characters. The field length is the distance on the input line from the position where the field begins to the next tab stop. The difference between the total length of all the substrings and the field length is incorporated as horizontal padding space that is divided among the indicated padding places. The incorporated padding is allowed to be negative. For example, if the field delimiter is # and the padding indicator is ^, then #^xxx^right# specifies a right-justified string with the string xxx centered in the remaining space.

Table 3.N is a summary and explanation of tab, leader, and field requests.

## 3.10 Input/Output Conventions and Character Translations

### 3.10.1 Input Character Translations

The newline character delimits input lines. In addition, STX, ETX, ENQ, ACK, and BEL are accepted and may be used as delimiters or translated into a graphic with a .tr request. All others are ignored.

The escape character (\) introduces sequences that cause the following character to mean another character or to indicate some function. A complete list of such sequences is given in Table 3.B. The escape character:

- should not be confused with the ASCII control character ESC of the same name
- can be input with the sequence \\
- can be changed with .ec, and all that has been said about the default \ becomes true for the new escape character.

A \e sequence can be used to print the current escape character. If necessary or convenient, the escape mechanism may be turned off with .eo and restored with .ec. A summary and explanation of input character translations requests are contained in Table 3.O.

### 3.10.2 Ligatures

Five ligatures are available in the **troff** character set: fi, fl, ff, ffi, and ffl. They may be input (even in the **nroff** formatter) by \fi, \fl, \ff, \ffi, and \ffl, respectively. The ligature mode is normally on in the **troff** formatter and automatically invokes ligatures during input. A summary and explanation of ligature requests are included in Table 3.O.

### 3.10.3 Backspacing, Underlining, and Overstriking

Unless in copy mode, the ASCII backspace character is replaced by a backward horizontal motion having the width of the space character. Underlining as a form of line drawing and, as a generalized overstriking function, is described in Part 12.

The **nroff** processor underlines characters automatically in the underline font, specifically with the .uf request. The underline font is normally on font position 2 (Times Italic). In addition to .ft request and \fF escape sequence, the underline font may be selected by .ul and .cu requests. Underlining is restricted to an output-device-dependent subset of reasonable characters. A summary and explanation of backspacing, underlining, and overstriking requests are included in Table 3.O.

### 3.10.4 Control Characters

Both the *break* control character (.) and the (*no-break*) control character ' may be changed, if desired. Such a change must be compatible with the design of any macros used in the span of the change and particularly of



any trap-invoked macros. A summary and explanation of the .cc and .c2 control character requests are included in Table 3.0.

### 3.10.5 Output Translation

One character can be made a stand-in for another character using the .tr request. All text processing (e.g., character comparisons) takes place with the input (stand-in) character which appears to have the width of the final character. Graphic translation occurs at the moment of output (including diversion). Included in Table 3.0 is a summary and explanation of the output translation request.

### 3.10.6 Transparent Throughput

An input line beginning with a \! is read in copy mode and transparently output (without the initial \!); the text processor is otherwise unaware of the line's presence. This mechanism may be used to pass control information to a post-processor or to imbed control lines in a macro created by a diversion.

### 3.10.7 Comments and Concealed Newline Characters

An uncomfortably long input line that must stay one line (e.g., a string definition or no-filled text) can be split into many physical lines by ending all but the last one with the escape \. The sequence \<newline> is ignored except in a comment. Comments may be imbedded at the end of any line by prefacing them with \". The newline character at the end of a comment cannot be concealed. A line beginning with \" will appear as a blank line and behave like .sp 1; a comment can be on a line by itself by beginning the line with \. \".

## 3.11 Local Horizontal/Vertical Motion and Width Function

### 3.11.1 Local Motion

The functions \v'N' and \h'N' can be used for local vertical and horizontal motion, respectively. The distance N may be negative; the positive directions are *rightward* and *downward*. A local motion is one contained within a line. To avoid unexpected vertical dislocations, it is necessary that the net vertical local motion (within a word in filled text and otherwise within a line) balance to zero. The above and certain other escape sequences providing local motion are summarized and explained in Table 3.P. As an example, E<sup>2</sup> is generated by the sequence E\v'-0.5'\s-4\&2\s0\v'0.5'.

### 3.11.2 Width Function

The width function \w'string' generates the numerical width of *string* (in basic units). Size and font changes may be imbedded in *string* and will not affect the current environment. For example, .ti-\w'1.'u could be used to temporarily indent leftward a distance equal to the size of the string "1".

The width function also sets three number registers. The registers st and sb are set respectively to the highest and lowest extent of *string* relative to the baseline; then, for example, the total height of the string is \n(stu-\n(sbu. In the troff formatter, the number register ct is set to a value between 0 and 3:

- 0 means that all characters in *string* are short lowercase characters without descenders (like e)
- 1 means that at least one character has a descender (like y)
- 2 means that at least one character is tall (like H)
- 3 means that both tall characters and characters with descenders are present.



### 3.11.3 Mark Horizontal Place

The escape sequence `\kx` will cause the current horizontal position in the input line to be stored in register `x`. As an example, the construction `\kx word \h' \nxu+2u' word` will embolden *word* by backing up almost to its beginning and overprinting it, resulting in **word**.

### 3.12 Overstrike, Zero-Width, Bracket, and Line Drawing Functions

#### 3.12.1 Overstrike

Automatically centered overstriking of up to nine characters is provided by the overstrike function

`\o'string'`.

Characters in *string* are overprinted with centers aligned; the total width is that of the widest character. The *string* should not contain local vertical motion. As examples, `\o'e\` produces *é*, and `\o'>/` produces *➤*.

#### 3.12.2 Zero-Width Characters

The function `\zc` will output *c* without spacing over it and can be used to produce left-aligned overstruck combinations. As examples, `\z\ci\pl` will produce  $\oplus$ , and `\(br\z\rfn\ul\br` will produce the smallest possible constructed box.

#### 3.12.3 Large Brackets

The Special Mathematical Font contains a number of bracket construction pieces that can be combined into various bracket styles. The function `\b'string'` may be used to pile up vertically the characters in *string* (the first character on top and the last at the bottom); the characters are vertically separated by one em and the total pile is centered one-half em above the current base line (one-half line in the `nroff` formatter). For example:

`\b\lc\lf'E\b\rc\rf'x'-0.5m'x'0.5m'`

produces:

$\left[ E \right]$

#### 3.12.4 Line Drawing

The function `\l'Nc'` will draw a string of repeated *c*'s toward the right for a distance *N* (*l* is lowercase L).

- If *c* looks like a continuation of an expression for *N*, it may be insulated from *N* with a `\&`.
- If *c* is not specified, the base-line rule (`_`) is used (underline character in `nroff`).
- If *N* is negative, a backward horizontal motion of size *N* is made before drawing the string.

Any space resulting from *N*/(size of *c*) having a remainder is put at the beginning (left end) of the string. In the case of characters that are designed to be connected, such as base-line rule (`_`), underrule (`\ul`), and root



en ( $\backslash$ (ru), the remainder space is covered by overlapping. If  $N$  is less than the width of  $c$ , a single  $c$  is centered on a distance  $N$ . As an example, a macro to underscore a string can be written

```
.de us
\l'$1\l'0\ul'
```

or one to draw a box around a string:

```
.de bx
\br\l'$1\l'0\br\l'0\l'0\ul'
```

such that

```
.us "underlined words"
```

and

```
.bx "words in a box"
```

yield

underlined words

and

words in a box

The function  $\backslash L'Nc$  will draw a vertical line consisting of the optional character  $c$  stacked vertically apart one em (one line in  $nroff$ ), with the first two characters overlapped, if necessary, to form a continuous line. The default character is box rule ( $\backslash$ (br); the other suitable character is bold vertical ( $\backslash$ (bv). The line is begun without any initial motion relative to the current base line. A positive  $N$  specifies a line drawn downward, and a negative  $N$  specifies a line drawn upward. After the line is drawn, no compensating motions are made; the instantaneous base line is at the end of the line.

The horizontal and vertical line drawing functions may be used in combination to produce large boxes. The zero-width *box-rule* and the one-half em wide *underrule* were designed to form corners when using one em vertical spacings. For example, the macro

```
.de eb
.sp -1 \ " compensate for next automatic base-line spacing
.nf \ " avoid possibly overflowing word buffer
\h'-.5n\l'\l'nau-1\l'\l'n(.lu+1n\l'\l'-\l'nau+1\l'0u-.5n\
\ul' \ " draw box
.fi
```



will draw a box around some text whose beginning vertical place was saved in number register *a* (e.g., using `.mk a`).

### 3.13 Hyphenation

The automatic hyphenation may be switched off and on. When switched on with `.hy`, several variants may be set. A hyphenation indicator character may be imbedded in a word to specify desired hyphenation points or may be prepended to suppress hyphenation. In addition, the user may specify a small exception word list. The default condition of hyphenation is off.

Only words that consist of a central alphabetic string surrounded by nonalphabetic strings (usually null) are considered candidates for automatic hyphenation. Words that were input containing hyphens (minus), em-dashes (`\(em)`), or hyphenation indicator characters (such as mother-in-law) are always subject to splitting after those characters whether or not automatic hyphenation is on or off. Table 3.Q is a summary and explanation of hyphenation requests.

### 3.14 Three-Part Titles

The titling function `.tl` provides for automatic placement of three fields at the left, center, and right of a line with a title length specifiable with `.lt`. The `.tl` may be used anywhere and is independent of the normal text collecting process. A common use is in header and footer macros. Table 3.R is a summary and explanation of 3-part title requests.

### 3.15 Output Line Numbering

Automatic sequence numbering of output lines may be requested with `.nm`. When in effect, a 3-digit, Arabic number plus a digit-space is prepended to output text lines. Text lines are offset by four digit-spaces and otherwise retain their line length. A reduction in line length may be desired to keep the right margin aligned with an earlier margin. Blank lines, other vertical spaces, and lines generated by `.tl` are not numbered. Numbering can be temporarily suspended with `.nn` or with a `.nm` followed by a later `.nm +0`. In addition, a line number indent *I* and the number-text separation *S* may be specified in digit-spaces. Further, it can be specified that only those line numbers that are multiples of some number *M* are to be printed (the others will appear as blank number fields). Table 3.S is a summary and explanation of output line numbering requests.

Figure 3.2 is an example of output line numbering. Paragraph portions are numbered with  $M=3$ :

- `.nm 1 3` was placed at the beginning;
- `.nm +0` was placed in front of the second paragraph;
- and `.nm` was placed at the end.

Line lengths were also changed (by `\w'0000'u`) to keep the right side aligned. Another example is `.nm +5 5 x 3`, which turns on numbering with the line number of the next line to be five greater than the last numbered line, with  $M=5$ , spacing *S* untouched, and the indent *I* set to 3.

### 3.16 Conditional Acceptance of Input

In Table 3.T, which is a summary and explanation of conditional acceptance requests:

- *c* is a 1-character, built-in condition name.
- `!` signifies *not*.
- *N* is a numerical expression.



- *string1* and *string2* are strings delimited by any nonblank, nonnumeric character not in the strings.
- *anything* represents what is conditionally accepted.

Built-in condition names are:

| CONDITION NAME | TRUE IF                     |
|----------------|-----------------------------|
| o              | Current page number is odd  |
| e              | Current page number is even |
| t              | Formatter is <b>troff</b>   |
| n              | Formatter is <b>nroff</b>   |

If condition *c* is true, if number *N* is greater than zero, or if strings compare identically (including motions and character size and font), *anything* is accepted as input. If a **!** precedes the condition, number, or string comparison, the sense of the acceptance is reversed.

Any spaces between the condition and the beginning of *anything* are skipped over. The *anything* can be either a single input line (text, macro, or whatever) or a number of input lines. In the multiline case, the first line must begin with a left delimiter **\{** and the last line must end with a right delimiter **\}**.

The request **.ie** (if-else) is identical to **.if** except that the acceptance state is remembered. A subsequent and matching **.el** (else) request then uses the reverse sense of that state. The **.ie**—**.el** pairs may be nested. For example:

```
.if e .tl ' Even Page %'
```

outputs a title if the page number is even, and

```
.ie\n%>1\{\
.sp 0.5i
.ti 'Page %'
.sp1.2i\}
.el .sp2.5i
```

treats page 1 differently from other pages.

### 3.17 Environment Switching

A number of parameters that control text processing are gathered together into an environment, which can be switched by the user. Environment parameters are those associated with some requests. The tables at the end of this section indicate in the "Explanation" column those requests so affected. In addition, partially collected lines and words are in the environment. Everything else is global; examples are page-oriented parameters, diversion-oriented parameters, number registers, and macro and string definitions. All environments are initialized with default parameter values. Table 3.U is a summary and explanation of the environment switching request.

### 3.18 Insertions From Standard Input

The input can be switched temporarily to the system standard input with **.rd** and switched back when two newline characters in a row are found (the extra blank line is not used). This mechanism is intended for insertions in form-letter-like documentation. On the UNIX operating system, the standard input can be the user keyboard, a pipe, or a file.



If insertions are to be taken from the terminal keyboard while output is being printed on the terminal, the command line option `-q` will turn off the echoing of keyboard input and prompt only with BEL. The regular input and insertion input cannot simultaneously come from the standard input. As an example, multiple copies of a form letter may be prepared by entering insertions for all copies in one file to be used as the standard input and causing the file containing the letter to reinvok itself by using the `.nx` request. The process would be ended by a `.ex` request in the insertion file.

Table 3.V is a summary and explanation of insertions from the standard input requests.

### 3.19 Input/Output File Switching

Table 3.W is a summary and explanation of input/output file switching requests.

### 3.20 Miscellaneous

Table 3.X is a summary and explanation of miscellaneous requests.

### 3.21 Output and Error Messages

Output from `.tm`, `.pm`, and prompt from `.rd`, as well as various error messages are written onto the UNIX operating system standard message output. The latter is different from the standard output, when compared to the `nroff` formatted output. By default, both are written onto the user's terminal, but they can be independently redirected.

Various error conditions may occur during the operation of the `nroff` and `troff` formatters. Certain less serious errors having only local impact do not cause processing to terminate. Two examples are:

- *word overflow*—caused by a word that is too large to fit into the word buffer (in fill mode)
- *line overflow*—caused by an output line that grew too large to fit in the line buffer.

In both cases, a message is printed, the offending excess is discarded, and the affected word or line is marked at the point of truncation with a `*` (in `nroff`) or a `<right hand>` (in `troff`). The philosophy is to continue processing, if possible, on the grounds that output useful for debugging may be produced. If a serious error occurs, processing terminates, and an appropriate message is printed. Examples are the inability to create, read, or write files, and the exceeding of certain internal limits that make future output unlikely to be useful.

Table 3.Y is a summary and explanation of output and error messages requests.

### 3.22 Compacted Macros

The time required to read a macro package by the `nroff` formatter may be lessened by using a compacted macro (a preprocessed version of a macro package). The compacted version is equivalent to the noncompacted version, except that a compacted macro package cannot be read by the `.so` request. A compacted version of a macro package, called *name*, is used by the `-cname` command line option, while the uncompact version is used by the `-mname` option. Because `-cname` defaults to `-mname` if the *name* macro package has not been compacted, the user should always use `-c` rather than `-m`.

#### 3.22.1 Building a Compacted Macro Package

Only macro, string, and diversion definitions; number register definitions and values; environment settings; and trap settings can be compacted. End macro (`em`) requests and any commands that may interact during



package interpretation with command-line settings (such as references in the MM macro package to the number register P, which can be set from the command line) are not compactible. There are two steps to make a compacted macro from a macro package:

- Separate compactible from noncompactible parts
- Place noncompactible material at the end of the macro package with a `.co` request. The `.co` request indicates to the `nroff` formatter when to compact its current internal state.

Compactable Material

`.co`

Noncompactible Material

### 3.22.2 Produce Compacted Files

When compactible and noncompactible segments have been established, the `nroff` formatter may be run with the `-k` option to build the compacted files. For example, if the output file to be produced is called `mac`, the following may be used to build the compacted files:

```
nroff -kmac mac
```

This command causes the `nroff` formatter to create two files in the current directory, `d.mac` and `t.mac`. The macro file must contain a `.co` request. Only lines before the `.co` request will be compacted. Both `-k` and `.co` are necessary. If no `.co` is found in the file, the `-k` is ignored. Likewise, if no `-k` appears on the command line, the `.co` is ignored.

Each macro package must be compacted separately by the `nroff` formatter. Compacted macro packages depend on the particular version of the `nroff` formatter that produced them. Any compacted macro packages must be recompactd when a new version of an `nroff` formatter is installed. If it is discovered that a macro package was produced by a different version than that attempting to read it, the `-c` will be abandoned, and the equivalent `-m` option attempted instead.

### 3.22.3 Install Compacted Files

The two compacted files, `d.mac` and `t.mac`, must be installed in the system macro library (`/usr/lib/macros`) with the proper names. If the files were produced by an `nroff` formatter, `cmp.n.` must be prepended to their names. For example, if the macro package is called `mac`, the two `nroff` formatter compacted files may be installed by

```
cp d.mac /usr/lib/macros/cmp.n.d.mac
```

or

```
cp t.mac /usr/lib/macros/cmp.n.t.mac
```

### 3.22.4 Install Noncompactible Segment

The noncompactible segment from the original macro package must be installed on the system as

```
/usr/lib/macros/ucmp.[nt].mac
```



where **n** of **[nt]** means the **nroff** formatter version, and **t** means the **troff** formatter version. The noncompactible segment must be produced manually by using the editor. Using the **mac** package as an example, the following could be used to install the **nroff** formatter noncompactible segment:

```
$ ed mac
/^\.co$/+,$w /usr/lib/macros/ucmp.n.mac
```

#### 4. TROFF Tutorial

##### 4.1 Overview

An important rule of using the **troff** formatter is to use it through an intermediary. In many ways the **troff** formatter resembles an assembly language, remarkably powerful and flexible, but nonetheless such that many operations must be specified at a level of detail and in a form that is too difficult for most people to use effectively.

There are programs that provide an interface to the **troff** formatter for the majority of users for two special applications.

- The **eqn** program provides an easy to learn language for typesetting mathematics. The user does not need to know the **troff** formatter to typeset mathematics.
- The **tbl** program provides the same convenience for producing tables of arbitrary complexity.

For producing text that may contain mathematics or tables, there are a number of macro packages that define formatting rules and operations for specific styles of documents and reduce the amount of direct contact with the **troff** formatter. In particular, the Memorandum Macros (MM) package provides most of the facilities needed for a wide range of document preparation. There are also packages for viewgraphs and other special applications. These packages are easier to use than the **troff** formatter once the user gets beyond the most trivial operations. They should be considered first.

In the few cases where existing packages do not accomplish the job, the solution is not to write an entirely new set of **troff** instructions from scratch but to make small changes to adapt packages that already exist. In accordance with this philosophy, the part of the **troff** formatter described here is only a small part of the whole, although it tries to concentrate on the more useful parts. The emphasis is on doing simple things and making incremental changes to what already exists. The **troff** formatter described is the C language version running on the UNIX operating system at Murray Hill.

To use the **troff** formatter, the actual text must be prepared plus some information that describes how it is to be printed. Text and formatting information are intimately intertwined. Most commands to the **troff** formatter are placed on a line separate from the text itself, one command per line beginning with a period. For example

```
Some text.
.ps 14
Some more text.
```

will change the point size of the letters being printed to 14 point (one point is 1/72 of an inch).

Occasionally, something special occurs in the middle of a line, such as an exponent. The formula for the area of a circle is typed as follows:

```
Area = \(*p\fr\fr\s8\u2\d\s0
```

The backslash character (\) is used to introduce **troff** commands and special characters within a line of text.



#### 4.2 Point Sizes and Line Spacing

The `.ps` request sets the point size. Since one point is 1/72 inch, 6-point characters are 1/12 inch high, and 36-point characters are 1/2 inch high. There are 15 point sizes—6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, and 36 point. Point size is rounded up to the next valid value, with a maximum of 36, if the number following the `.ps` request is not a legal value.

If no number follows the `.ps` request, point size reverts to the previous value. The troff processor begins with point size 10. Point size can also be changed in the middle of a line or a word with a `\s` escape sequence. The `\s` sequence should be followed by a legal point size. The `\s0` sequence causes the size to revert to its previous value. The `\s1011` sequence can be understood correctly as “size 10, followed by an 11”. Caution should be exercised with similar constructions.

Relative size changes are also legal and useful:

```
\s-2UNCLE\s+2
```

temporarily decreases the size by two points, then restores it. Relative size changes have the advantage that the size difference is independent of the starting size of the document. The amount of the relative change is restricted to a single digit.

Another parameter that determines what the type looks like is the spacing between lines. It is set independently of the point size. Vertical spacing is measured from the bottom of one line to the bottom of the next. The command to control vertical spacing is `.vs`. For running text, it is usually best to set the vertical spacing about 20 percent larger than the character size. For example, a usable combination would be

```
.ps 9
.vs 11p
```

Vertical spacing is partly a matter of taste, depending on how much text is to be squeezed into a given space, and partly a matter of traditional printing style. By default, the troff formatter uses a point size of 10 and a vertical spacing of 12. When `.vs` is used without arguments, vertical spacing reverts to the previous value.

The `.sp` request is used to get extra vertical space. Used alone, it gives one extra blank line (whatever `.vs` is set). Since that may be more or less than desired, `.sp` can be followed by information about how much space is wanted. For instance:

```
.sp 1.5i      means “a space of 1.5 inches” (most troff processor installations understand decimal frac-
              tions)
.sp 2i        means “two inches of vertical space”
.sp 2p        means “two points of vertical space”
.sp 2 or .sp 2v means “two vertical spaces” (two of whatever .vs is set).
```

These same scale factors can be used after the `.vs` request to define line spacing. Scale factors can be used after most commands that deal with physical dimensions.

All size numbers are converted internally to *machine units*, which are 1/432 inch (1/6 point). For most purposes, this is enough resolution to provide good accuracy of representation. The situation is not quite so good vertically, where resolution is 1/144 inch (1/2 point).



### 4.3 Fonts and Special Characters

The troff processor and the typesetter allow four different fonts at one time. Normally, three fonts (Times Roman, Times Italic, and Times Bold) and one collection of special characters are permanently mounted. The Greek, mathematical symbols, and miscellany of the special font are listed in Table 3.D.

The troff processor prints in Roman unless otherwise commanded. To change the font, the .ft request is used:

|       |                          |
|-------|--------------------------|
| .ft B | switch to bold font.     |
| .ft I | switch to italics font.  |
| .ft R | switch to Roman font.    |
| .ft P | return to previous font. |
| .ft   | return to previous font. |

The underline request (.ul) causes the next input line to print in italics. It can be followed by a count to indicate that more than one line is to be italicized.

Fonts can also be changed within a line or word with the \f in-line sequences. For instance

**boldface text**

is produced by

\fBbold\fIface\fR text

If it is desired to do this so the previous font is left undisturbed, extra \fP sequences should be inserted:

\fBbold\fP\fIface\fP\fR text\fP

Since only the immediately previous font is remembered, the previous font must be restored after each change or it will be lost. The same is true of .ps and .vs when used without an argument.

There are other fonts available besides the standard set, although only four can be used at any given time. The .fp request tells the troff formatter what fonts are actually mounted on the typesetter. For example:

.fp 3 H

says that the Helvetica font is mounted on position 3. A list of fonts and what they look like are shown in Fig. 3.1. Appropriate .fp requests should appear at the beginning of a document if standard fonts are not used.

It is possible to make a document relatively independent of the actual fonts used to print it by using font numbers instead of names. For example: \f3 and .ft3 mean "whatever font is mounted at position 3". Normal settings are Roman font on 1, italic on 2, bold on 3, and special on 4.

There is also a way to get synthetic bold fonts by overstriking letters with a slight offset. The .bd request addresses this function.

Special characters have 4-character input names beginning with \ ( and may be inserted anywhere in the text. In particular, Greek letters are all of the form \(\*-, where - is an uppercase or lowercase Roman font letter reminiscent of the Greek. A list of these special names is given in Table 3.D.

Some characters are automatically translated into others: grave (`) and acute (') accents become open and close single quotation marks. Similarly, a typed minus sign becomes a hyphen. The \- input will print an explicit minus sign. A \e entry causes a backslash to be printed.



#### 4.4 Indents and Line Lengths

The troff processor starts with a line length of 6.5 inches, which is too wide for 8-½ inch by 11-inch paper. The .ll request resets the line length. For example:

```
.ll 6i
```

As with the .sp request, the actual length can be specified in several ways; inches are probably the most intuitive. The maximum line length provided by the typesetter is 7.5 inches. To use the full width, the default physical left margin (page offset) must be reset. This is done by the .po request. The margin is normally slightly less than 1 inch from the left edge of the paper. The .po 0 request sets the offset as far to the left as it will go.

The indent request (.in) causes the left margin to be indented by some specified amount from the page offset. If .in is used to move the left margin to the right and the .ll is used to move the right margin to the left, offset blocks of text are obtained. As an example

```
.in 0.5i
.ll -0.5i
text to be set into a block
.ll +0.5i
.in -0.5i
```

will create a block that looks like:

A clergyman at Cambridge preached a sermon which one of his auditors commended. "Yes," said a gentleman to whom it was mentioned, "it was a good sermon, but he stole it." This was told to the preacher. He resented it, and called on the gentleman to retract what he had said. "I am not," replied the aggressor, "very apt to retract my words, but in this instance I will. I said, you had stolen the sermon; I find I was wrong; for on returning home, and referring to the book whence I thought it was taken, I found it there."

The use of + and - changes the previous setting by the specified amount rather than just overriding it. The distinction is quite important:

- .ll +1i makes lines 1 inch longer
- .ll 1i makes lines 1 inch long.

With the .in, .ll, and .po requests, the previous value is used if no argument is specified.

The .ti request is used to temporarily indent a single line. For example, all paragraphs in this manual effectively begin with the .ti 3 request. Since no units are specified, the line is indented three ems by default. The default unit for .ti, as for most horizontally oriented requests (.ll, .in, .po), is ems. An em is roughly the width of the letter m in the current point size. Precisely, an em in size *p* is *p* points. Although inches are usually clearer than ems to people who do not set type for a living, ems have a place: they are a measure of size that is proportional to the current point size. The ems unit is used to make text that keeps its proportions regardless of point size. The ems can be specified as scale factors directly, as in .ti 2.5m.



Lines can be indented negatively if the indent is already positive:

```
.ti -3i
```

causes the next line to be moved back 3/10 of an inch.

To make a decorative initial capital that is three lines high:

- The whole paragraph is indented.
- The initial character is moved back with the .ti request.
- The initial character is made bigger (e.g., \s36N\s0) and moved down from its normal position (see Part 6).

#### 4.5 Tabs

Tabs (the ASCII **horizontal tab** character) can be used to produce output in columns or to set the horizontal position of output. Typically, tabs are used only in unfilled text. Tab stops are set by default every half inch from the current indent but can be changed by the .ta request. Tab stops are set every inch, for example, with the following entry:

```
.ta 1i 2i 3i 4i 5i 6i
```

Tab stops are left justified (as on a typewriter), so lining up columns of right-justified numbers can be a problem. If there are many numbers or if a table layout is needed, the tbl program is available (Section 4).

A handful of numeric columns can be done by preceding every number with enough blanks to make it line up when typed. For instance:

```
.nf
.ta 1i 2i 3i
  1tab 2tab 3
 40tab 50tab 60
 700tab 800tab 900
.fi
```

Each leading blank is a \0 string. This is a character that does not print but has the same width as a digit. When printed it produces:

```
  1    2    3
 40   50   60
700  800  900
```



It is also possible to fill up tabbed-over space with some character other than blanks by setting the tab replacement character with the `.tc` request:

```
.ta 1.5i 2.5i
.tc \ (ru      \" (ru is _)
Name tab Age tab
```

produces

```
Name_____ Age _____
```

To reset the tab replacement character to a blank, the `.tc` request (with no argument) is used. Lines can also be drawn with the `\l` escape sequence as described in paragraph 4.6.4.

The `troff` processor provides a general mechanism called "fields" for setting up complicated columns. This is used by the `tbl` program.

#### 4.6 Local Motions

The `troff` processor provides a number of escape sequences for placing characters of any size at any place. They can be used to draw special characters or to tune the output for a particular appearance. Most of these sequences are straightforward but messy to read and tough to type correctly.

##### 4.6.1 Vertical Motions

If the `eqn` program is not used, subscripts and superscripts are most easily done with the half-line local motions `\u` and `\d` sequences. To go back up the page half a point size, a `\u` is inserted at the desired place; to go down half a point size, a `\d` is inserted. The `\u` and `\d` should always be used in pairs. Since `\u` and `\d` refer to the current point size, they should either be both inside or both outside the size changes. Otherwise, an unbalanced vertical motion will result.

Sometimes the space given by `\u` and `\d` is not the right amount. The `\v` sequence can be used to request an arbitrary amount of vertical motion. The in-line sequence `\v' N` causes motion up or down the page by the amount specified in `N`. For example, to move the character "P" down, the following would apply:

```
.in +0.6i      (indent paragraph)
.ll -0.3i      (shorten lines)
.ti -0.3i      (move N back)
\v'2'\s36N\s0\v'-2'ott met Shott, Nott
shot at Shott. . .
```

A minus sign causes upward motion, while no sign or a plus sign means down the page. Thus `\v'-2'` causes an upward vertical motion of two line spaces.

There are many other ways to specify the amount of motion

```
\v'0.1i'
\v'3p'
\v'-0.5m'
```

etc. are all legal. The scale specifier `i`, `p`, or `m` goes inside the quotes. Any character can be used in place of the quotes. This is true of all other `troff` formatter commands and sequences described in this section.

Since the `troff` formatter does not take within-the-line vertical motions into account when figuring where it is on the page, output lines can have unexpected positions if the left and right ends are not at the same vertical



position. Thus `\v`, like `\u` and `\d`, should always balance upward vertical motion in a line with the same amount in the downward direction.

#### 4.6.2 Horizontal Motions

Arbitrary horizontal motions are also available, `\h` is analogous to `\v`, except that the default scale factor is ems instead of line spaces. As an example,

```
\h'-0.1i'
```

causes a backwards motion of a tenth of an inch. In a practical situation, when printing the mathematical symbol  $>>$ , the default spacing is too wide, so `eqn` replaces this by

```
>\h'-0.3m'>
```

to produce  $>>$ .

Frequently, `\h` is used with the "width function" `\w` to generate motions equal to the width of some character string. The construction `\w'thing'` is a number equal to the width of **thing** in machine units (1/432 inch). All **troff** formatter computations are ultimately done in these units. To move horizontally, the width of an **xfR**, `\h' \w'x'u'` is used. Since the default scale factor for all horizontal dimensions is **m** (ems), **u** (machine units) must be used, or the motion produced will be too large. Nested quotes are acceptable to the **troff** formatter as long as none are omitted. An example of this kind of construction would be to print the string **.sp** by overstriking with a slight offset. The following example puts out **.sp**, moves left by the width of **.sp**, moves right one unit, and prints **.sp** again:

```
.sp\h'-\w'.sp'u'\h'lu'.sp
```

Part 11 describes a way of avoiding typing so much input for each command name.

There are several special-purpose **troff** formatter sequences for local motion:

- The `\O` is an unpaddingable (never widened or split across a line-by-line justification and filling) white space of the same width as a digit.
- The `\<space>` is an unpaddingable character the width of a space.
- The `\|` is 1/6 the width of a space.
- The `\^` is 1/12 the width of a space.
- The `\&` has zero width and is useful in entering a text line that would otherwise begin with a ..
- The `\o` sequence causes up to nine characters to be overstruck, centered on the widest. This is for accents such as:

```
syst\o" e\ga" me t\o" e\aa" l\o" e\aa" phonique
```

which produces

```
système téléphonique
```

The accents `\(ga` and `\(aa` (`\'` and `\'`) are just one character to the **troff** formatter.



#### 4.6.3 Overstrikes

Overstrikes can be made with another special convention, `\z`, the zero-motion sequence. Normal horizontal motion is suppressed with the `\zx` after printing the single character `x`, so another character can be laid on top of it. Although sizes can be changed within `\o`, characters are centered on the widest, and there can be no horizontal or vertical motions. The `\z` may be the only way to get what is needed.

A more ornate overstrike is given by the bracketing function `\b`, which piles up characters vertically, centered on the current base line. Thus big brackets are obtained by constructing them with piled-up smaller pieces.

#### 4.6.4 Drawing Lines

A convenient facility for drawing horizontal and vertical lines of arbitrary length with arbitrary characters is provided by the `troff` formatter. A 1-inch long line is printed with a `\l'li'` sequence. The length can be followed by the character to use if the `_` is not appropriate. The `\l'0.5i.'` sequence draws a 1/2-inch line of dots. Escape sequence `\L` is analogous, except that it draws a vertical instead of a horizontal line. The document titled "Table Formatting Program" describes other ways of providing horizontal and vertical lines.

#### 4.7 Strings

If a paper contains a large number of occurrences of an acute accent over a letter `e`, typing `\o "e\'"` for each `é` would be a nuisance. Fortunately, the `troff` formatter provides a way to store an arbitrary collection of text in a "string", and thereafter use the string name as a shorthand for its contents. Strings are one of several `troff` formatter mechanisms whose judicious use permits typing a document with less effort and organizing it so that extensive format changes can be made with few editing changes.

A reference to a string is replaced by whatever text as defined the string. Strings are defined with the `.ds` request. The line

```
.ds e \o "e\'"
```

defines the string `e` to have the value `\o "e\'"`.

String names may be either 1 or 2 characters long. They are referred to by `\*x` for 1-character names or `\*(xy` for 2-character names. Thus, to get

```
téléphone
```

given the definition of the string `e` as above,

```
t\*el\*ephone
```

is the input.

If a string must begin with blanks, it is defined as

```
.ds xx "      text
```

The double quote signals the beginning of the definition. There is no trailing quote; the end of the line terminates the string.

A string may be several lines long. If the `troff` formatter encounters a `\` at the end of any line, it is thrown away and the next line is added to the current one. A long string can be made by ending each line except the last with a backslash.



```
.ds xx this \
is a very \
long string
```

Strings may be defined in terms of other strings or even in terms of themselves.

#### 4.8 Introduction to Macros

In its simplest form, a macro is a shorthand notation similar to a string. For instance, if every paragraph is to start in exactly the same way, with a space and a temporary indent of two ems, the following requests would perform the operation:

```
.sp
.ti +2m
```

To save typing these requests every time used, they could be collapsed into one shorthand line, such as a **troff** command, **.PP**. The **.PP** is called a *macro*. The way to tell the **troff** formatter what **.PP** means is to define it with the **.de** request:

```
.de PP
.sp
.ti +2m
..
```

The first line names the macro (**.PP** in this example). It is in uppercase so it will not conflict with any name that the **troff** formatter might already know about. The last line (**..**) marks the end of the definition. In between is the text which is inserted whenever the **troff** formatter sees the **.PP** macro call. A macro can contain any mixture of text and formatting requests.

The definition of a macro has to precede its first use; undefined macros are ignored. Names are restricted to one or two characters.

Using macros for commonly occurring sequences of requests is important since it saves typing and makes later changes easier. If it is decided that in producing a document the paragraph indent is too small, the vertical space is too large, and Roman font should be forced, only the definition of **.PP** needs to be changed to read

```
.de PP          \" paragraph macro
.sp 2p
.ti +3m
.ft R
..
```

The change takes effect everywhere **.PP** is used and is easier than changing commands throughout the whole document.

A **troff** formatter escape sequence that causes the rest of the line to be ignored is **\**. It is used to add comments to the macro definition (a wise idea once definitions get complicated).

Another example of macros that start and end a block of offset, unfilled text is

```
.de OS          \" start indented block
.sp
.nf
.in +0.3i
```



```

..
.de OE          \" end indented block
.sp
.fi
.in -0.3i
..

```

The .OS and .OE macros could be used before and after text to provide the following effect:

```

Copy to
John Doe
Richard Roberts
Stanley Smith

```

In this example, the indentation used is .in +0.3i instead of .in 0.3i. This permits the nesting of the .OS and .OE macros to get blocks within blocks.

Should the amount of indentation be changed at a later date, it is necessary to change only the definitions of .OS and .OE, not individual requests throughout the whole paper.

#### 4.9 Titles, Pages, and Numbering

Titles, pages, and numbering is a complicated area where nothing is done automatically. Of necessity, some of this section is a cookbook to be copied literally until some experience is obtained.

To get a title at the top of each page, such as:

```

left top          center top          right top

```

it was possible on an older system (**roff**) to get headers and footers automatically on every page with the following:

```

.he 'left top'center top'right top'
.fo 'left bottom'center bottom'right bottom'

```

This does not work in the **troff** formatter. Instead, specifications must be provided:

- What to do at and around the title line
- When to print the title
- What the actual title is.

The .NP macro (new page) is defined to process titles at the end of one page and the beginning of the next:

```

.de NP
'bp
'sp 0.5i
.tl 'left top'center top'right top'
'sp 0.3i
..

```

These requests are explained as follows:

- The 'bp (begin page) command causes a skip to the top-of-page.



- The `'sp 0.5i` command will space down  $\frac{1}{2}$  inch.
- The `.tl` command prints the title.
- The `'sp 0.3i` provides another 0.3 inch space.

The reason that the `'bp` and `'sp` commands are used instead of the `.bp` and `.sp` requests is that the `.sp` and `.bp` cause a break to take place. This means that all the input text collected but not yet printed is flushed out as soon as possible, and the next input line is guaranteed to start a new line of output. Had `.bp` been used in the `.NP` macro, a break in the middle of the current output line would occur when a new page is started. The effect would be to print the left-over part of that line at the top of the page, followed by the next input line on a new output line. This is not desired. Using `'` instead of `.` for a command tells the **troff** formatter that no break is to take place. The output line currently being filled should not be forced out before the space or new page.

The list of requests that cause a break is short and natural:

```
.bp .br .ce .fi .nf .sp .in .ti
```

Other requests cause no break, regardless of whether a `.` or a `'` is used. If a break is really needed, a `.br` request at the appropriate place will provide it.

To ask for `.NP` at the bottom of each page, a statement like "when the text is within an inch of the bottom of the page, start the processing for a new page" is used. This is done with the `.wh` request. For example:

```
.wh -1i NP
```

No `.` character is used before `NP` since it is simply the name of a macro and not a macro call. The minus sign means "measure up from the bottom of the page", so `-1i` means 1 inch from the bottom.

The `.wh` request appears in the input outside the definition of `.NP`. Typically, the input would be

```
.de NP
--- body of macro
..
.wh -1i NP
```

As text is actually being output, the **troff** formatter keeps track of its vertical position on the page; and after a line is printed within 1 inch from the bottom, the `.NP` macro is activated.

- The `.wh` request sets a trap at the specified place.
- The trap is sprung when that point is passed.

The `.NP` macro causes a skip to the top of the next page (that is what the `'bp` was for) and prints the title with appropriate margins.

Something to beware of when changing fonts or point sizes is crossing a page boundary in an unexpected font or size. Titles come out in the size and font most recently specified instead of what was intended. The length of a title is independent of the current line length, so titles will come out at the default length of 6.5 inches unless changed. Changing title length is done with the `.lt` request.

There are several ways to fix the problems of point sizes and fonts in titles. The `.NP` macro can be changed to set the proper size and font for the title, and then restore the previous values, like this:

```
.de NP
```



```

'bp
'sp 0.5i
.ft R          \" set title font to Roman
.ps 10         \" set size to 10 point
.lt 6i         \" set length to 6 inches
.tl 'left'center'right'
.ps           \" revert to previous size
.ft P         \" and to previous font
'sp 0.3i

```

This version of **.NP** does not work if the fields in the **.tl** request contain size or font changes. To cope with that contingency requires the **troff** formatter "environment" mechanism discussed in Part 13.

To get a footer at the bottom of a page, the **.NP** macro should be modified. One option is to have the **.NP** macro do some processing before the **'bp** request. Another option is to split the macro into a footer macro (invoked at the bottom margin) and a header macro (invoked at the top of page).

Output page numbers are computed automatically as each page is produced (starting at 1), but no numbers are printed unless explicitly requested. To get page numbers printed, the **%** character should be included in the **.tl** request at the position where the number is to appear. For example:

```
.tl "- % -"
```

centers the page number inside hyphens. The page number can be set at any time with either a **.bp n** request (which immediately starts a new page numbered **n**) or with **.pn n** (which sets the page number for the next page but does not cause a skip to the new page). The **.bp +n** sets the page number to **n** more than its current value. The **.bp** request without an argument means **.bp +1**.

#### 4.10 Number Registers and Arithmetic

The **troff** processor has a facility for doing arithmetic and defining and using variables with numeric values, called *number registers*. Number registers, like strings and macros, can be useful in setting up a document so it is easy to change later. They also serve for any sort of arithmetic computation.

Like strings, number registers have 1- or 2-character names. They are set by the **.nr** request and are referenced anywhere by **\nx** (1-character name) or **\n(xy)** (2-character name).

There are quite a few predefined number registers maintained by the **troff** formatter, among them:

- **%** for the current page number
- **nl** for the current vertical position on the page
- **dy**, **mo**, and **yr** for the current day, month, and year
- **.s** and **.f** for the current size and font (the font is a number from one to four).

Any of these can be used in computations like any other register, but some, like **.s** and **.f**, cannot be changed with **.nr**.

An example of the use of number registers is in an older macro package where most significant parameters are defined in terms of the values of a handful of number registers. These include the point size for text, the vertical spacing, and the line and title lengths. To set the point size and vertical spacing, a user may input



```
.nr PS 9
.nr VS 11
```

The paragraph macro, **.PP**, is roughly defined as follows:

```
.de PP
.ps \n(PS      \" reset size
.vs \n(VSp     \" spacing
.ft R          \" font
.sp 0.5v       \" half a line
.ti +3m
..
```

This sets the font to Roman and the point size and line spacing to whatever values are stored in the number registers **PS** and **VS**.

The reason for two backslashes is to indicate that a backslash is really meant. When the **troff** formatter originally reads the macro definition, it peels off one backslash to see what is coming next. Two backslashes in the definition are required to ensure that a backslash is left in the definition when the macro is used. If only one backslash is used, point size and vertical spacing will be frozen at the time the macro is defined, not when it is used.

Protecting by an extra layer of backslashes is needed only for **\n**, **\\***, **\\$**, and **\** itself. Things like **\s**, **\f**, **\h**, **\v**, etc. do not need an extra backslash since they are converted by the **troff** formatter to an internal code immediately upon detection.

Arithmetic expressions can appear anywhere that a number is expected. As an example:

```
.nr PS \n(PS-2
```

decrements **PS** by 2. Expressions can use the arithmetic operators **+**, **-**, **\***, **/**, **%** (mod), the relational operators **>**, **>=**, **<**, **<=**, **=**, **!=** (not equal), and parentheses.

So far, the arithmetic has been straightforward; more complicated things are tricky.

- Number registers hold only integers. In the **troff** formatter, arithmetic uses truncating integer division just like Fortran.
- In the absence of parentheses, evaluation is done left-to-right without any operator precedence including relational operators. Thus:

```
7*-4+3/13
```

becomes -1.

Number registers can occur anywhere in an expression and so can scale indicators like **p**, **i**, **m**, etc. (but no spaces). Although integer division causes truncation, each number and its scale indicator is converted to machine units (1/432 inch) before any arithmetic is done, so **1i/2u** evaluates to **0.5i** correctly.

The scale indicator **u** often has to appear when least expected, in particular when arithmetic is being done in a context that implies horizontal or vertical dimensions. For example, **.ll 7/2i** is not 3-1/2 inches. Instead, it is really 7 ems/2 inches. When translated into machine units, it becomes 0 (this is because the default units for



horizontal parameters (like .ll) are ems). Another incorrect try is .ll 7i/2. The 2 is 2 ems, so 7i/2 is small, although not 0. The .ll 7i/2u must be used. A safe rule is to attach a scale indicator to every number, even constants.

For arithmetic done within a .nr request, there is no implication of horizontal or vertical dimension, so the default units are "units", and 7i/2 and 7i/2u mean the same thing. Thus:

```
.nr 11 7i/2
```

```
.ll \n(llu
```

accomplishes what is desired as long as the u on the .ll request is included.

#### 4.11 Macros With Arguments

Two things are needed to be able to define macros that can change from one use to the next according to parameters supplied as arguments:

1. When the macro is defined, it must be indicated that some parts will be provided as arguments when the macro is called.
2. When the macro is called, the actual arguments to be plugged into the definition must be provided.

An example would be to define a macro (.SM) that will print its argument two points smaller than the surrounding text.

```
.de SM
```

```
\s-2\\$1\s+2
```

```
..
```

The macro call would appear:

```
.SM SMALL
```

The argument SMALL in this example would then appear two points smaller than the rest of the print.

Within a macro definition, the symbol \\\$n refers to the nth argument with which the macro was called. Thus \\\$1 is the string to be placed in a smaller point size when .SM is called.

A slightly more complicated version is the following definition of .SM which permits optional second and third arguments that will be printed in the normal size:

```
.de SM
```

```
\\$3\s-2\\$1\s+2\\$2
```

```
..
```

Arguments not provided when the macro is called are treated as empty. The macro call

```
.SM ABLE ),
```

would appear (with ABLE in smaller type)

```
ABLE),
```

The macro call

```
.SM BAKER ). (
```



produces the following (with BAKER in smaller print):

(BAKER).

It is convenient to reverse the order of arguments because trailing punctuation is much more common than leading. The number of arguments that a macro was called with is available in number register `.$`.

The macro, `.BD`, is used to make "bold Roman" for `troff` formatter command names in text. It combines horizontal motions, width computations, and argument rearrangement:

```
.de BD
\&\\$3\fl\\$1\h'-\w'\\$1'u+2u'\\$1\fp\\$2
..
```

The `\h` and `\w` escape sequences need no extra backslash. The `\&` is there in case the argument begins with a period. Two backslashes are needed with the `\\$n` commands to protect one of them when the macro is being defined. A second example will make this clearer. A `.SH` macro can be defined to produce automatically numbered section headings with the title in smaller size bold print. The use is

```
.SH "Section title ..."
```

If the argument to a macro is to contain blanks, it must be surrounded by double quotes.

The definition of the `.SH` macro is

```
.nr SH 0          \ " initialize section number
.de SH
.sp 0.3i
.ft B
.nr SH \\n(SH+1   \ " increment number
.ps \\n(PS-1      \ " decrease PS number
\\n(SH.  \\$1      \ " title
.ps \\n(PS        \ " restore PS
.sp 0.3i
.ft R
..
```

The section number is kept in number register `SH`, which is incremented each time just before use.

**Note:** A number register may have the same name as a macro without conflict but a string may not.

A `\\n(SH` and `\\n(PS` was used instead of a `\n(SH` and `\n(PS`. Had `\n(SH` been used, it would have yielded the value of the register at the time the macro was defined, not at the time it was used. Similarly, by using `\\n(PS`, the point size at the time the macro was called is obtained.

An example that does not involve numbers is the `.NP` macro (defined earlier) which had the request

```
.tl 'left'center'right'
```

The fields could be made into parameters by using instead

```
.tl '\*(LT'\*(CT'\*(RT'
```

The title comes from three strings called `LT`, `CT`, and `RT`. If these are empty, the title will be a blank line. Normally, `CT` would be set with

```
.ds CT - % -
```



to give just the page number between hyphens. A user could supply private definitions for any of the strings.

#### 4.12 Conditionals

Suppose it is desired that the **.SH** macro leave two extra inches of space just before Section 1, but nowhere else. The cleanest way to do that is to test inside the **.SH** macro whether the section number is 1, and add some space if it is. The **.if** command provides the conditional test that can be added just before the heading line is output:

```
.if \n(SH=1 .sp 2i \ " first section only
```

The condition after the **.if** request can be any arithmetic or logical expression. If the condition is logically true or arithmetically greater than zero, the rest of the line is treated as if it were text (a request in this case). If the condition is false, zero, or negative, the rest of the line is skipped.

It is possible to do more than one request if a condition is true. For example, if several operations are to be done prior to Section 1, the **.S1** macro is defined and invoked when Section 1 is almost complete (as determined by an **.if**).

```
.de S1
--- processing for section 1
```

```
.de SH
```

```
---
.if \n(SH=1 .S1
```

```
..
```

An alternate way is to use the extended form of the **.if** request, e.g.:

```
.if \n(SH=1 \{--- processing
for section 1 ---\}
```

The braces, **\{** and **\}**, must occur in the positions shown or unexpected extra lines will be in the output. The **troff** processor also provides an "if-else" construction.

A condition can be negated by preceding it with **!**. The same effect as above is obtained (but less clearly) by using

```
.if !\n(SH>1 .S1
```

There are a handful of other conditions that can be tested with **.if**. For example:

```
.if e .tl 'left top' center top 'right top'
.if o .tl 'left top' center top 'right top'
```

gives facing pages different titles, depending on whether the page number is even or odd, when used inside an appropriate new page macro.

Two other conditions are **t** and **n**, which tells whether the formatter is **troff** or **nroff**:

```
.if t troff stuff ...
```



.if n nroff stuff ...

String comparisons may be made in a .if request.

.if 'string1'string2' stuff

executes the program **stuff** if *string1* is the same as *string2*. The character separating the strings can be anything reasonable that is not contained in either string. The strings themselves can reference strings with \\*, arguments with \\$, etc.

#### 4.13 Environments

There is a potential problem when going across a page boundary: parameters like *size* and *font for a page title* may be different from those in effect in the text when the page boundary occurs. A general way to deal with this and similar situations is provided by the **troff** formatter.

There are three environments. Each has independently selectable versions of many parameters associated with processing, including size, font, line and title lengths, fill/no-fill mode, tab stops, and partially collected lines. Thus the titling problem may be solved by processing the main text in one environment and titles in another with its own suitable parameters.

The **.ev n** request shifts to environment **n** (**n** must be 0, 1, or 2). The **.ev** request with no argument returns to the previous environment. Environment names are maintained in a stack, so calls for different environments may be nested and unwound consistently.

If the main text is processed in environment 0 where the **troff** formatter begins by default, the *new page* macro, **.NP**, can then be modified to process titles in environment 1, e.g.

```
.de NP
.ev 1          \" shift to new environment
.lt 6i        \" set parameters here
.ft R
.ps 10
    --- any other processing
.ev           \" return to previous environment
```

It is also possible to initialize the parameters for an environment outside the **.NP** macro, but the version shown keeps all the processing in one place and is easier to understand and change.

#### 4.14 Diversions

There are numerous occasions in page layout when it is necessary to store some text for a period of time without actually printing it. Footnotes are the most obvious example. Text of the footnote usually appears in the input well before the place on the page is reached where it is to be printed. The place where it is output normally depends upon the magnitude of the footnote. This implies that there must be a way to process the footnote, at least enough to decide its size without printing it.

A mechanism called a diversion is provided by the **troff** formatter for doing this processing. Any part of the output may be diverted into a macro instead of being printed; and at some convenient time, the macro may be put back into the input.

The **.di xy** request begins a diversion. All subsequent output is collected into the macro **xy** until the **.di** request with no arguments is encountered. This terminates the diversion. Processed text is available at any time



thereafter by giving the `.xy` request. The vertical size of the last finished diversion is contained in the built-in number register `dn`. For instance, to implement a keep-release operation so that text between the macros `.KS` and `.KE` will not be split across a page boundary (as for a figure or table), the following applies:

- When a `.KS` is encountered, the output is diverted to determine its size.
- When a `.KE` is encountered and if the diverted text will fit on the current page, it is printed there. If the diverted text does not fit on the current page, it is printed at the top of the next page.

The definitions of the `.KS` and `.KE` macros are as follows:

```
.de KS          \" start keep
.br            \" start fresh line
.ev 1          \" collect in new environment
.fi           \" make it filled text
.di XX        \" collect in XX

.de KE          \" end keep
.br           \" get last partial line
.di           \" end diversion
.if \\n(dn>=\\n(.t .bp \" bp if does not fit
.nf           \" bring it back in no-fill
.XX           \" text
.ev           \" return to normal environment
```

The number register `nl` indicates the current position on the output page. Since output was being diverted, it remains at its value when the diversion started. The `dn` register contains the amount of text in the diversion. The distance to the next trap is in the built-in register `.t`. It is assumed that the next trap is at the bottom margin of the page. If the diversion is large enough to go past the trap, the `.if` is satisfied; and a `.bp` request is issued. In either case, the diverted output is brought back with `.XX`. It is essential to bring it back in no-fill mode so the `troff` formatter will do no further processing on it.

This is not the most general keep-release operation nor is it robust in the face of all conceivable inputs. It would require more space than available to display it in full generality. This manual is not intended to teach everything about diversions, but to sketch out enough so that existing macro packages can be read with some comprehension.

## 5. NROFF/TROFF Tutorial Examples

Although the `nroff` and `troff` formatters have by design a syntax reminiscent of earlier text processors with the intent of easing their use, it is usually necessary to prepare at least a small set of macro definitions to describe most documents. Such common formatting needs such as page margins and footnotes are deliberately not built into the `nroff` and `troff` formatters. Instead, the macro and string definition, number register, diversion, environment switching, page-position trap, and conditional input mechanisms provide the basis for user-defined implementations.

Examples in the following text are intended to be useful and somewhat realistic but will not necessarily cover all relevant contingencies. Explicit numerical parameters are used to make the examples easier to read and to illustrate typical values. In many cases, number registers would be used to reduce the number of places where numerical information is kept and to concentrate conditional parameter initialization data that depends on whether the `troff` or `nroff` formatter is being used.

### 5.1 Page Margins

Header and footer macros are defined to describe the top and bottom page margin areas, respectively. A trap is planted at page position 0 for the header and at  $-N$  ( $N$  from the page bottom) for the footer. A simple header and footer macro definition is



```
.de hd      \" define header
'sp li
..          \" end definition
.de fo      \" define footer
'bp
..          \" end definition
.wh 0 hd
.wh -1i fo
```

This example provides blank 1-inch top and bottom margins. The header will occur on the first page, only if the definition and trap exist prior to the initial pseudopage transition. In fill mode, the output line that springs the footer trap was typically forced out because some part or whole word did not fit on it. If anything in the footer and header that follows causes a break, that word or part word will be forced out. In this and other examples, requests like **bp** and **sp** that normally cause breaks are invoked using the *no-break* control character ('). When the header/footer design contains material requiring independent text processing, the environment may be switched to avoid interaction with running text.

A more realistic example follows:

```
.in +4
.de hd      \" header
.if t .tl \"\r\n\" \r\n' \" troff cut mark
.if \\n%>1
'sp 10.5i-1 \" tl base at 0.5i
.tl \"- % -\" \" centered page number
.ps        \" restore size
.ft        \" restore font
.vs {}     \" restore vs
'sp 11.0i  \" space to 1.0i
.ns        \" turn on no-space mode
..
.de fo      \" footer
.ps 10     \" set footer/header size
.ft R      \" set font
.vs 12p    \" set base-line spacing
.if \\n%=1
'sp 1\\n(.pu-0.5i-1 \" tl base 0.5i up
.tl \"- % -\" \" first page number
'bp
..
.wh 0 hd
.wh -1i fo
```

This example sets the size, font, and base-line spacing parameters for the header/footer material. Parameters are restored to their original values when the header or footer is completed. The material in this case is a page number at the bottom of the first page and at the top of the remaining pages. If the **troff** formatter is used, a cut mark is drawn in the form of *root-en's* at each margin. The **sp's** refer to absolute positions to avoid dependence on the base-line spacing. Another reason for the **sp** in the footer is that the footer is invoked by printing a line whose vertical spacing swept past the trap position by possibly as much as the base-line spacing. The *no-space* mode is turned on at the end of **hd** to render ineffective accidental occurrences of **sp** at the top of the running text.

The above method of restoring size, font, etc. presupposes that such requests (that set *previous* value) are not used in the running text. A better scheme is to save and to restore both the current and previous values



as shown for size in the following:

```
.de fo
.nr s1 \\n(.s  \" current size
.ps
.nr s2 \\n(.s  \" previous size
---          \" rest of footer
..
.de hd
---          \" header stuff
.ps \\n(s2    \" restore previous size
ps \\n(s1     \" restore current size
..
```

Page numbers may be printed in the bottom margin by a separate macro triggered during the footer's page ejection:

```
.de bn          \" bottom number
.tl \"- % -\"    \" centered page number
..
.wh -0.5i-1v bn \" tl base 0.5i up
```

## 5.2 Paragraphs and Headings

Housekeeping associated with starting a new paragraph should be collected in a paragraph macro that does the desired preparagraph spacing, forces the correct font, size, base-line spacing, and indent; checks that enough space remains for more than one line; and requests a temporary indent.

```
.de pg          \" paragraph
.br            \" break
.ft R          \" force font,
.ps 10         \" size,
.vs 12p        \" spacing,
.in 0          \" and indent
.sp 0.4        \" prespace
.ne 1+\\n(.Vu    \" want more than 1 line
.ti 0.2i       \" temp indent
..
```

The first break in **pg** will force out any previous partial lines and must occur before the **.vs** request. The forcing of font, size, base-line spacing, and indent is partly a defense against prior error and partly to permit things like section heading macros to set parameters only once. The prespacing parameter is suitable for the **troff** formatter; a larger space, at least as big as the output device vertical resolution, would be more suitable in the **nroff** formatter. The choice of remaining space to test for in the **.ne** is the smallest amount greater than one line (the **.V** is the available vertical resolution).

A macro to automatically number section headings might look like:

```
.de sc          \" section
---            \" force font, etc.
.sp 0.4        \" prespace
.ne 2.4+\\n(.Vu  \" want 2.4+ lines
.fi
\\n+S.
..
.nr S 0 1      \" initial S
```

The usage is **sc**, followed by the section heading text, followed by **pg**. The **.ne** test value includes one line of heading, 0.4 line in the following **pg**, and one line of the paragraph text. A word consisting of the next section



number and a period is produced to begin the heading line. The format of the number may be set by the `.af` request.

Another common form is the labeled, indented paragraph where the label protrudes left into the indent space.

```
.de lp          \" labeled paragraph
.pg
.in 0.5i        \" paragraph indent
.ta 0.2i 0.5i   \" label, paragraph
.ti 0
\t\\$\\1\\t\\c \" flow into paragraph
..
```

The intended usage is

```
.lp label
```

The *label* will begin at 0.2 inch and cannot exceed a length of 0.3 inch without intruding into the paragraph. The label could be right adjusted against 0.4 inch by setting the tabs instead with `.ta 0.4iR 0.5i`. The last line of `lp` ends with `\\c` so that it will become a part of the first line of the text that follows.

### 5.3 Multiple Column Output

The production of multiple column pages requires the footer macro to decide whether it was invoked by other than the last column, so that it will begin a new column rather than produce the bottom margin. The header can initialize a column register that the footer will increment and test. The following is arranged for two columns but is easily modified for more:

```
.de hd          \" header
---
.nr cl 0 1      \" init column count
.mk            \" mark top of text
..
.de fo          \" footer
.ie \\n+(cl<2 \\ \"
.po +3.4i       \" next column; 3.1+0.3
.rt            \" back to mark
.ns \\          \" no-space mode
.el \\          \"
.po \\nMu       \" restore left margin
---
'bp \\
..
.ll 3.1i        \" column width
.nr M \\n(.o    \" save left margin
```

Typically, a portion of the top of the first page contains full width text; the request for the narrower line length, as well as another `.mk` request, will be made where the 2-column output is to begin.

### 5.4 Footnote Processing

The footnote mechanism is used by imbedding the footnotes in the input text at the point of reference demarcated by an initial `.fn` and a terminal `.ef`.



**.fn**  
Footnote text and control lines.  
**.ef**

The following macro definitions cause footnotes to be processed in a separate environment and diverted for later printing in the space immediately prior to the bottom margin. There is provision for the case where the last collected footnote does not completely fit in the available space:

```
.de hd          \" header
---
.nr x 0 1      \" init footnote count
.nr y 0--\\nb  \" current footer place
.ch fo --\\nbu  \" reset footer trap
.if \\n(dn .fz  \" leftover footnote
..
.de fo          \" footer
.nr dn 0       \" zero last diversion size
.if \\nx \\{
.ev 1          \" expand footnotes in ev1
.nf           \" retain vertical size
.FN           \" footnotes
.rm FN        \" delete it
.if \" \\n(.z \" fy \" .di  \" end overflow diversion
.nr x 0       \" disable fx
.ev          \" pop environment
---
'bp
..
.de fx          \" process footnote overflow
.if \\nx .di fy \" divert overflow
..
.de fn          \" start footnote
.da FN         \" divert (append) footnote
.ev 1          \" in environment 1
.if \\n+x=1 .fs \" if first, include separator
.fi           \" fill mode
..
.de ef          \" end footnote
.br           \" finish output
.nr z \\n(.v    \" save spacing
.ev           \" pop ev
.di           \" end diversion
.nr y --\\n(dn   \" new footer position
.if \\nx=1 .nr y - (\\n(.v--\\nz) \" uncertainty correction
.ch fo \\nyu     \" y is negative
.if (\\n(nl+lv)>(\\n(p+\\ny)\\  \"
.ch fo \\n(nlu+lv \" it did not fit
..
.de fs          \" separator
\\'1i'          \" 1 inch rule
.br
..
.de fz          \" get leftover footnote
.fn
```



```

.nf          \" retain vertical size
.fy          \" where fx put it
.ef

..
.nr b 1.0i   \" bottom margin size
.wh 0 hd     \" header trap
.wh 12i fo    \" footer trap, temp position
.wh -\\n bu fx \" fx at footer position
.ch fo -\\n bu \" conceal fx with fo

```

- The header macro (**hd**) initializes a footnote count register **x** and sets both the current footer trap position register **y** and the footer trap itself to a nominal position specified in register **b**.
- If the register **dn** indicates a leftover footnote, the **fz** macro is invoked to reprocess it.
- The footnote start macro (**fn**) begins a diversion (append) in environment 1 and increments the footnote count register **x**; if the count is one, the footnote separator macro (**fs**) is interpolated. The separator is kept in a separate macro to permit user redefinition.
- The footnote end macro (**ef**) restores the previous environment and ends the diversion after saving spacing size in register **z**.
- Register **y** is decremented by the size of the footnote which is available in register **dn**.
- On the first footnote, register **y** is further decremented by the difference in vertical base-line spacings of the two environments. This prevents late triggering of the footer trap from causing the last line of the combined footnotes to overflow.
- The footer trap is set to the lower of **y** or the current page position (**nl**) plus one line to allow for printing the reference line.
- If indicated by **x**, the footer **fo** rereads the footnotes from **FN** in no-fill mode in environment 1 and deletes **FN**. If the footnotes were too large to fit, the macro **fx** will be trap-invoked to redirect the overflow into **fy**, and the register **dn** will later indicate to the header whether or not **fy** is empty.
- Both **fo** and **fx** macros are planted in the nominal footer trap position in an order that causes **fx** to be concealed unless the **fo** trap is moved.
- The footer terminates the overflow diversion (if necessary) and zeros **x** to disable **fx**. This is because the uncertainty correction, together with a not-too-late triggering of the footer, can result in footnote macros finishing before reaching the **fx** trap.

### 5.5 Last Page

After the last input file has ended, **nroff** and **troff** formatters invoke the end macro, if any, and eject the remainder of the page.

```

.de en      \" end-macro
\c
"bp
..
.em en

```

During the eject, any traps encountered are processed normally. At the end of this last page, processing terminates unless a partial line, word, or partial word remains. If it is desired that another page be started, the end-macro will deposit a null partial word and effect another last page.



## TABLE FORMATTING PROGRAM

### 1. Introduction

The **tbl** program is a document formatting preprocessor for the formatter which makes fairly complex tables easy to specify and enter. Tables consist of columns which may be independently centered, right-adjusted, left-adjusted, or aligned by decimal points. Headings may be placed over single columns or groups of columns. A table entry may contain equations or consist of several rows of text. Horizontal or vertical lines may be drawn as desired in the table, and any table or element may be enclosed in a box.

A description of a table is put by the **tbl** program into a **troff** formatter (the **nroff** and/or **troff** processor will be referred to synonymously as "formatter") list of requests that prints the table. The **tbl** program isolates a portion of a job that can be successfully handled and leaves the remainder for other programs. Thus, **tbl** may be used with the equation formatting program (**eqn**) and/or various formatter layout macro packages without function duplication.

### 2. Usage

On the UNIX operating system, the **tbl** program can be run on a simple table with the command

```
tbl filename | troff
```

For more complicated use, where there are several input files containing equations and *ms* or *mm* macro requests as well as tables, the normal command is

```
tbl file1 file2 ... | eqn | troff -ms
```

The usual options may be used on the **troff** formatter and **eqn** commands. Usage of the **nroff** formatter is similar to that of **troff**, but only TELETYPE® Model 37 and Diablo-mechanism (DASI or GSI) terminals can print boxed tables. If a file name is "-", the standard input is read at that point.

For the convenience of users employing line printers without adequate driving tables or post-filters, there is a special **-TX** command-line option to **tbl** which produces output that does not have fractional line motions. The only other command-line options recognized by **tbl** are **-ms** and **-mm**. They are turned into commands to fetch the corresponding macro files. It is usually more convenient to place these arguments on the **troff** formatter part of the command line, but they are accepted by **tbl** as well.

When **eqn** and **tbl** programs are used together on the same file, **tbl** should be used first. If there are no equations within tables, either sequence works. It is usually faster to execute **tbl** first since **eqn** normally produces a larger expansion of the input. However, if there are equations within tables (using the *delim* option in **eqn**), **tbl** must be executed first or the output will be scrambled. Use of equations in *n*-style columns should be avoided since **tbl** attempts to split numerical format items into two parts. The *delim(xx)* table option prevents splitting numerical columns within delimiters. For example, if the **eqn** delimiters are **\$\$**, giving *delim(\$ \$)* causes a numerical column such as **1245 \$± 16\$** to be divided after 1245, not after 16.

The **tbl** program accepts up to 35 columns; the actual number that can be processed may be smaller depending on availability of **troff** formatter number registers. Number register names used by **tbl** must be avoided within tables. These include 2-digit numbers from 31 to 99 and strings of the form *4x*, *5x*, *#x*, *x+*, *x!*, *^x*, and *x-*, where *x* is any lowercase letter. The names **##**, **#-**, and **#^** are also used in certain circumstances. To conserve register names, the *n* and *a* key letters (key letters are introduced in the "Format Section" part that follows) share a register; hence, the restriction that they may not be used in the same column.

As an aid in writing layout macros, **tbl** defines a number register *TW* which is the table width. The *TW* number register is defined by the time the **.TE** macro is invoked and may be used in the expansion of that macro.



More importantly, to assist in laying out multipage boxed tables, the macro `T#` is defined to produce the bottom lines and side lines of a boxed table and then be invoked at its end. By use of this macro in the page footer, a multipage table can be boxed. In particular, the *ms* and *mm* macros can be used to print a multipage boxed table with a repeated heading by giving the argument *H* to the `.TS` macro. If the table start macro is written

```
.TS H
```

a line of the form

```
.TH
```

must be given in the table after any table heading (or at the start if none). Material up to the `.TH` is placed at the top of each page of the table. The remaining lines in the table are placed on several pages as required. This is not a feature of `tbl` but of the *ms* and *mm* macros.

### 3. Input Commands

Input to `tbl` is text for a document with tables preceded by a `.TS` (table start) command and followed by a `.TE` (table end) command. The `tbl` program processes the tables, generates formatting requests, and leaves the text unchanged. The `.TS` and `.TE` lines are copied so that `troff` formatter layout macros (such as memorandum formatting macros) can use these lines as delimiters. Arguments on the `.TS` or `.TE` lines are copied, but otherwise ignored, and may be used by document layout macro requests.

The format of the input is

```
text
.TS
table
.TE
text
.TS
table
.TE
text
...
```

The format of each table is

```
.TS
options;
format.
data
.TE
```

Each table is independent and contains:

- Global options
- A format section describing individual columns and rows of the table
- Data to be printed.

The format section and data are always required but not the options.



### 3.1 Global Options

There may be a single line of options affecting the whole table. If present, this line must immediately follow the .TS line and must contain a list of option names separated by spaces, tabs, or commas and must be terminated by a semicolon. Allowable options are:

- **center**—center table (default is left-adjust)
- **expand**—make table as wide as current line length
- **box**—enclose table in a box
- **allbox**—enclose each item of table in a box
- **doublebox**—enclose table in two boxes
- **tab (x)**—separate data items by using *x* instead of tab
- **linesize (n)**—set lines or rules (e.g., from **box**) in *n*-point type
- **delim (xy)**—recognize *x* and *y* as *eqn* delimiters.

The **tbl** program tries to keep boxed tables on one page by issuing appropriate **.ne** (need) requests. These requests are calculated from the number of lines in the tables. If there are spacing requests embedded in the input, the **.ne** requests may be inaccurate. Normal **troff** formatter procedures, such as keep-release macros, are used in that case. If a multipage boxed table is required, macros designed for this purpose (**.TS H** and **.TH**) should be used.

### 3.2 Format Section

The format section of the table specifies the layout of the columns. Each line in the format section corresponds to one line of table data (except that the last format line corresponds to all following data lines up to any additional **.T&** command line). Each line contains a **key letter** for each column of the table. Key letters for each column may be separated by spaces or tabs for readability purposes. Key letters are:

|        |                                                                                                                                                             |
|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| L or l | Indicates a left-adjusted column entry.                                                                                                                     |
| R or r | Indicates a right-adjusted column entry.                                                                                                                    |
| C or c | Indicates a centered column entry.                                                                                                                          |
| N or n | Indicates a numerical column entry. Numerical entries are aligned so that the units digits of numbers line up.                                              |
| A or a | Indicates an alphabetic subcolumn. All corresponding entries are aligned on the left and positioned so that the widest entry is centered within the column. |
| S or s | Indicates a spanned heading. The entry from the previous column continues across this column (not allowed for the first column of the table).               |
| ^      | Indicates a vertically spanned heading. The entry from the previous row continues down through this row (not allowed for the first row of the table).       |

When numerical column alignment (**n**) is specified, a location for the decimal point is sought. The rightmost dot (.) adjacent to a digit is used as a decimal point. If there is no dot adjoining a digit, the rightmost digit is



used as a units digit. If no alignment is indicated, the item is centered in the column. However, the special non-printing character string `\&` may be used to override dots and digits or to align alphabetic data. This string lines up where a dot normally would and then disappears from the final output. In the following example, items shown in the left column will be aligned (in a numerical column) as shown in the right column:

|           |         |
|-----------|---------|
| 13        | 13      |
| 4.2       | 4.2     |
| 26.4.12   | 26.4.12 |
| abcdefg   | abcdefg |
| abcdefg\& | abcdefg |
| 43\&3.22  | 433.22  |
| 749.12    | 749.12  |

If numerical data are used in the same column with wider L (the capital L key letter is used instead of lowercase for readability) or r-type table entries, the widest number is centered relative to the wider L or r items. Alignment within the numerical items is preserved. This is similar to the behavior of a-type data. Alphabetic subcolumns (requested by the a key letter) are always slightly indented relative to L items. If necessary, the column width is increased to force this. This is not true for n-type entries.

**Note:** The n and a items should not be used in the same column.

The end of the format section is indicated by a period. The layout of key letters in the format section resembles the layout of the actual data in the table. Thus, a simple 3-column format might appear as

```
css
l n n.
```

The first line of the table contains a heading centered across all three columns. Each remaining line contains a left-adjusted item in the first column followed by two columns of numerical data. A sample table in this format is:

| OVERALL TITLE |       |       |
|---------------|-------|-------|
| Item-a        | 34.22 | 9.1   |
| Item-b        | 12.65 | .02   |
| Items: c,d,e  | 23    | 5.8   |
| Total         | 69.87 | 14.92 |

Instead of listing the format of successive lines of a table on consecutive lines of the format section, successive line formats may be given on the same line, separated by commas. The format for the above example could be written:

```
c s s, l n n.
```

Additional features of the key letter system are:

- **Horizontal lines**—A key letter may be replaced by underscore (`_`) to indicate a horizontal line in place



of the column entry or equal (=) to indicate a double horizontal line. If an adjacent column contains a horizontal line or if there are vertical lines adjoining this column, the horizontal line is extended to meet nearby lines. If any data entry is provided for this column, it is ignored and a warning message is printed.

- *Vertical lines*—A vertical bar (|) placed between column key letters will cause a vertical line between the corresponding columns of the table. A vertical bar to the left of the first key letter or to the right of the last one produces a line at the edge of the table. If two vertical bars appear between key letters, a double vertical line is drawn.
- *Space between columns*—A number may follow the key letter indicating the amount of separation between this column and the next column. The number specifies the separation in *ens*. One *en* is about the width of the letter "n". More precisely, an *en* is the number of points (1 point = 1/72 inch) equal to half the current type size. If the *expand* option is used, these numbers are multiplied by a constant such that the table is as wide as the current line length. The default column separation number is 3. If the separation is changed, the worst case (largest space requested) governs.
- *Vertical spanning*—Vertically spanned items extending over several rows of the table are centered in their vertical range. If a key letter is followed by *t* or *T*, any corresponding vertically spanned item will begin at the top line of its range.
- *Font changes*—A key letter followed by a string containing a font name or number preceded by the letter *f* or *F* indicates that the corresponding column should be in a different font from the default font (usually Roman). All font names are one or two letters. A 1-letter font name should be separated from whatever follows by a space or tab. The single letters *B*, *b*, *I*, and *i* are shorter synonyms for *fB* and *fI*. Font-change requests given with the table entries override these specifications.
- *Point size changes*—A key letter followed by *p* or *P* and a number indicates the point size of the corresponding table entries. If the number is a signed digit, it is taken as an increment or decrement from the current point size. If both a point size and a column separation value are given, one or more blanks must separate them.
- *Vertical spacing changes*—A key letter followed by *v* or *V* and a number indicates the vertical line spacing used within a multiline table entry. The number may be a signed digit, in which case it is taken as an increment or decrement from the current vertical spacing. A column separation value must be separated by blanks or some other specification from a vertical spacing request. This request has no effect unless the corresponding table entry is a text block.
- *Column width indication*—A key letter followed by *w* or *W* and a width value in parentheses indicates minimum column width. If the largest element in the column is not as wide as the width value given after the *w*, the largest element is assumed to be that wide. If the largest element in the column is wider than the specified value, its width is used. The width is also used as a default line length for included text blocks. Normal *troff* formatter units can be used to scale the width value. The default value is *ens* if none are used. If the width specification is a unitless integer, the parentheses may be omitted. If another width value is given in a column, the last one controls the width.
- *Equal-width columns*—A key letter followed by *e* or *E* indicates equal-width columns. All columns whose key letters are followed by *e* or *E* are made the same width. This permits a group of regularly spaced columns.
- *Staggered columns*—A key letter followed by *u* or *U* indicates that the corresponding entry is to be moved up one-half line. This makes it easy to have a column of differences between numbers in an adjoining column. The *allbox* option does not work with staggered columns.
- *Zero-width item*—A key letter followed by *z* or *Z* indicates that the corresponding data item is to be ignored in calculating column widths. This may be useful in allowing headings to run across adjacent columns where spanned headings would be inappropriate.



- *Default*—Column descriptors missing from the end of a format line are assumed to be L. The longest line in the format section, however, defines the number of columns in the table. Extra columns in the data are ignored.

The order of the features is immaterial. They need not be separated by spaces except as indicated to avoid ambiguities involving point size and font changes. Thus, a numerical column entry in italic font and 12-point type with a minimum width of 2.5 inches and separated by 6 ens from the next column could be specified as

```
np12w(2.5i)fI 6
```

### 3.3 Data To Be Printed

Data for the table are input after the format section. Each table line is typed as one line of data. Very long input lines can be broken. Any line whose last character is a backslash (\) is combined with the following line (i.e., the \ vanishes). Data for different columns (table entries) are separated by tabs or by whatever character has been specified in the **tab** global option. There are a few special cases of data entries:

- *troff commands within tables*—An input line beginning with a dot and followed by anything but a number (.xx) is assumed to be a request to the formatter and is passed through unchanged, retaining its position in the table. For example, a space within a table may be produced with the .sp request in the data.
- *Full width horizontal lines*—An input line containing only the \_ (underscore) character or = (equal sign) is taken to be a single or double line, respectively, extending the full width of the table.
- *Single column horizontal lines*—An input table entry containing only the \_ character or the = is taken to be a single or double line extending the full width of the column. Such lines are extended to meet horizontal or vertical lines adjoining this column. To obtain these characters explicitly in a column, they should be preceded by a \& or followed by a space before the usual tab or newline character.
- *Short horizontal lines*—An input table entry containing only the string \\_ is assumed to be a single line as wide as the contents of the column. It is not extended to meet adjoining lines.
- *Repeated characters*—An input table entry containing only a string of the form \Rx, where x is any character, is replaced by repetitions of the character x as wide as data in the column. The sequence is not extended to meet adjoining columns.
- *Vertically spanned items*—An input table entry containing only the \^ character string indicates that the table entry immediately above spans downward over this row. It is equivalent to a table format key letter of ^.
- *Text blocks*—In order to include a block of text as a table entry, precede it by T{ and follow it by T}. Thus, the sequence

```
... T{
  block of
  text
  T} ...
```

is the way to enter as a single entry in the table something that cannot conveniently be typed as a simple string between tabs. The T} (end delimiter) must begin a line. Additional columns of data may follow after a tab on the same line.

Various limits in the **troff** program are likely to be exceeded if 30 or more text blocks are used in a table. This produces diagnostic messages such as "too many string/macro names" or "too many number registers".



Text blocks are pulled out from the table, processed separately by the formatter, and replaced in the table as a solid block.

If no line length is specified in the block of text or in the table format, the default is to use

$$L \times C / (N + 1)$$

where  $L$  is the current line length,  $C$  is the number of table columns spanned by the text, and  $N$  is the total number of columns in the table.

Other parameters (point size, font, etc.) used in setting the block of text are:

- (a) Those in effect at the beginning of the table (including the effect of the .TS macro)
- (b) Any table format specifications of size, spacing, and font using the *p*, *v*, and *f* modifiers to the column key letters
- (c) **troff** requests within the text block itself (requests within the table data but not within the text block do not affect that block).

Although any number of lines may be present in a table, only the first 200 lines are used in setting up the table. A multipage table may be arranged as several single-page tables if this proves to be a problem.

When calculating column widths, all table entries are assumed to be in the font and size being used when the .TS command was encountered. This is true except for font and size changes indicated in the table format section or within the table data (as in the entry `\s+3 data\fp\s0`). Because arbitrary **troff** requests may be sprinkled in a table, care must be taken to avoid confusing width calculations. It is not possible to change the number of columns, the space between columns, the global options such as *box*, or the selection of columns to be made equal in width.

#### 4. Additional Command Lines

To change the format of a table after many similar lines, as with subheadings or summarizations, the .T& (table continue) command is used to change column parameters. It is not recognized after the first 200 lines of a table. The outline of such a table input is

```
.TS
options;
format.
data
...
.T&
format.
data
.T&
format.
data
.TE
```

Using this procedure, each table can be close to its corresponding format line.

#### 5. Examples

Figures 3.3 through 3.8 are included to show input and output information that illustrates the basic concepts of the `tbl` program. The `T` symbol in the input data represents a tab character. Although each figure has a title



that indicates an option or feature, other examples of use may be gleaned from them. For instance, Fig. 3.7 also indicates the requesting of bold type print in the format area.



## MATHEMATICS TYPESETTING PROGRAM

### 1. Introduction

Mathematical text is known in the publishing trade as "penalty copy" because it is slower, more difficult, and more expensive to set in type than any other kind of copy normally occurring in books and journals. One difficulty is the multiplicity of characters, sizes, and fonts. Many mathematical expressions require an intimate mixture of Roman, italic, and Greek letters (in three sizes) and a number of special characters. Typesetting such expressions by traditional methods is essentially a manual operation.

A second difficulty is the 2-dimensional character of mathematics. This is illustrated by expressions such as:

$$\int \frac{dx}{ae^{mx} - be^{-mx}} = \begin{cases} \frac{1}{2m\sqrt{ab}} \log \frac{\sqrt{a}e^{mx} - \sqrt{b}}{\sqrt{a}e^{mx} + \sqrt{b}} \\ \frac{1}{m\sqrt{ab}} \tanh^{-1}\left(\frac{\sqrt{a}}{\sqrt{b}}e^{mx}\right) \\ \frac{-1}{m\sqrt{ab}} \coth^{-1}\left(\frac{\sqrt{a}}{\sqrt{b}}e^{mx}\right) \end{cases}$$

This example also shows line-drawing, built-up characters (such as braces and radicals), and a spectrum of positioning problems.

The **eqn** software for typesetting mathematics has been designed to be easy to learn and to use by people (for example, secretaries and mathematical typists) who know neither mathematics nor typesetting. The language can be learned in an hour or so since it has few rules and fewer exceptions. It interfaces directly with the phototypesetting language, the **troff** formatter, so mathematical expressions can be embedded in the running text of a manuscript, and the entire document produced in one process. Typical mathematical expressions include size and font changes, positioning, line drawing, and other necessary functions to print according to mathematical conventions, and are done automatically.

The syntax of the language is specified by a small context-free grammar; a compiler-compiler is used to make a compiler that translates this language into typesetting commands. Output may be produced on either a phototypesetter or on a terminal with forward and reverse half-line motions. So that mixtures of text and mathematics may be handled, the **eqn** program interfaces directly with text formatting programs.

### 2. Usage

On the UNIX operating system, the phototypesetter is driven by a text formatting program, **troff**, which was designed for typesetting text. Facilities needed for printing mathematical expressions, such as arbitrary horizontal and vertical motions, line drawing, and font size changing are also provided. Syntax for describing these special operations is difficult to learn and difficult even for experienced users to type correctly. For this reason, the **troff** formatter is used as an assembly language by the **eqn** program which describes and compiles mathematical expressions.

The **eqn** program will also produce mathematics on DASI and GSI terminals and on TELETYPE® Model 37 terminals. Input is identical, but **neqn** and the **nroff** formatter are used instead of **eqn** and the **troff** formatter. Some things will not look as good because terminals do not provide the variety of characters, sizes, and fonts that a typesetter does, but the output is usually adequate for proofreading.

Running a preprocessor is easy on the UNIX operating system. To typeset text stored in *files*, the following command is issued:

```
eqnfiles | troff
```



The vertical bar connects the output of one **eqn** process to the input of another **troff** process. Any **troff** formatter options are located following the **troff** formatter part of the command. For example:

```
eqn files | troff -ms
```

A compatible version of **eqn** can be used on devices like TELETYPE® Model 37, DASI, and GSI terminals which have half-line forward and reverse capabilities. To print equations on a TELETYPE® Model 37, the following command is used:

```
neqn files | nroff
```

The language for equations recognized by **neqn** is identical to that of **eqn** although the output is more restricted. To use a GSI or DASI terminal as the output device, the following command is used:

```
neqn files | nroff -Tx
```

where *x* is the terminal type being used, such as 300 or 300S.

The **eqn** and **neqn** programs can be used with the **tbl** program for typesetting tables that contain mathematics.

```
tbl files | eqn | troff
```

```
tbl files | neqn | nroff
```

### 3. Language

#### 3.1 Design

The fundamental principle upon which the **eqn** language design is based is that the language should be easy to use by those who know neither mathematics nor typesetting. This principle implies:

- Normal mathematical conventions about operator precedence, such as parentheses, cannot be used. To give special meaning to such characters means that the user has to understand what is being typed. The language should not assume that parentheses are always balanced.
- There should be few rules, keywords, special symbols, and operators. This keeps the language easy to learn and remember. Furthermore, there should be few exceptions to the rules that do exist. If something works in one situation, it should work everywhere. If a variable can have a subscript, then a subscript can have a subscript, etc., without limit.
- Standard things should happen automatically. When " $x=y+z+1$ " is typed, " $x=y+z+1$ " should be the result. Subscripts and superscripts should be printed automatically (with no special intervention) in appropriately smaller size. Fraction bars should be made the right length and positioned at the correct height. A mechanism for overriding default actions should exist, but its application is the exception, not the rule.

A secondary, but still important, design goal is that the system should be easy to build and to change. To this end and to guarantee regularity, the language is defined by a context-free grammar. The compiler for the language was built using a compiler-compiler.

The typist should have a reasonable picture (a 2-dimensional representation) of the desired final form, such as might be handwritten by the author of a paper. It is also assumed that the input is to be typed on a computer



terminal much like an ordinary typewriter. This implies an input alphabet of perhaps 100 characters, none of them special.

The **troff** processor performs work for the mathematics typesetting function. It is a powerful program, with a macro facility, text and arithmetic variables, numerical computation and testing, and conditional branching. Text strings are passed to the **troff** formatter omitting the need for a separate storage management package. The user need not be concerned with most details of the particular device and character set currently in use. For example, the **troff** formatter computes the widths of all strings of characters; the user does not need to know about them.

### 3.2 Structure

The basic structure of the language is not original. Equations are pictured as a set of boxes, pieced together in various ways. For example, something with a subscript is a box followed by another box moved downward and shrunk an appropriate amount. A fraction is a box centered above another box, at the right altitude, with a line of correct length drawn between them.

### 3.3 Mode of Operation

Since the **eqn** program is useful for typesetting mathematics only, it interfaces with the underlying typesetting language in order to get intermingled mathematics and text. The standard mode of operation is that when a document is typed, mathematical expressions are input as part of the text but marked by delimiters, **.EQ** and **.EN**. The program reads this input and treats as comments those things which are not mathematics, passing them through untouched. At the same time, it converts mathematical inputs into **troff** formatter commands. The resulting output is passed directly to the formatter where comments and mathematical parts become text and/or formatter commands.

## 4. User's Guide

### 4.1 Delimiters

The **eqn** preprocessor reads intermixed text and equations and passes its output to the **troff** formatter. Since the formatter uses lines beginning with a period as control words (**.ce** means "center the next output line"), **eqn** uses the **.EQ** macro to mark the beginning of an equation and the **.EN** macro to mark the end. The **.EQ** and **.EN** delimiters are passed through to the formatter untouched, so they can be used to center equations, number them automatically, etc. The **troff** and **nroff** formatter macro packages, **-ms** and **-mm**, allow equations to be centered, indented, left-justified, and numbered. The **-ms** package centers (by default) equations. To left-justify an equation, the **.EQ L** macro is used. A **.EQ I** macro will indent the equation. Any of these sequences can be followed by an arbitrary equation number placed at the right margin. For example, the input

```
.EQ I (3.1a)
x = f(y/2) + y/2
.EN
```

produces the output

$$x = f(y/2) + y/2 \quad (3.1a)$$

By default, however, **.EQ** and **.EN** are ignored by the **troff** formatter, so equations are printed in-line.

The **.EQ** and **.EN** macros can be supplemented by **troff** commands as desired. A centered display equation can be produced with the input

```
.ce
```



```
.EQ
x sub i = y sub i ...
.EN
```

Since it is tedious to type `.EQ` and `.EN` around very short expressions (e.g., single letters), two characters can be defined to serve as the left and right delimiters of expressions. These characters are recognized anywhere in subsequent text. For example, if the left and right delimiters have both been set to `#`, the input

Let `#x#`, `#y#`, and `#z#` be positive

produces

Let  $x$ ,  $y$ , and  $z$  be positive

## 4.2 Spaces and New Lines

### 4.2.1 Input Spaces

Input is free form. Space and newline characters in the input are used by `eqn` to separate pieces of the input; they are not used to create space in the output. Thus an input

```
x      =      y
      + z + 1
```

produces

$$x=y+z+1$$

Free-form input is easier to type initially. Space and newline characters should be freely used to make input equations readable and easy to edit. Very long lines are hard to correct if a mistake is made.

### 4.2.2 Output Spaces

Extra white space can be forced into the output by several characters of various sizes. A tilde (`~`) gives a space equal to the normal word spacing in text, a circumflex (`^`) gives half this much, and a tab character spaces to the next tab stop (tab stops must be set by `troff` commands). Spaces (or tildes, etc.) also serve to delimit pieces of input. For example, to get

$$x = y + z$$

the following expression is input

$$x\text{~}=\text{~}y\text{~}+\text{~}z$$

## 4.3 Symbols, Special Names, and Greek Alphabet

Mathematical symbols, mathematical names, and the Greek alphabet are known by `eqn`. For example:

$$x=2\pi\int\sin(\omega t)dt$$

produces



$$x = 2\pi \int \sin(\omega t) dt$$

Spaces in the input are necessary to indicate that *sin*, *pi*, *int*, and *omega* are separate entities and should get special treatment. The **eqn** program looks up each string of characters in a table, and if found, gives it a translation. Digits, parentheses, brackets, punctuation marks, and the following mathematical words are converted to Roman font:

sin cos tan sinh cosh tanh arc  
max min lim log ln exp  
Re Im and if for det

In the previous example, *pi* and *omega* become their Greek equivalents, *int* becomes the integral sign (which must be moved down and enlarged), and *sin* is output in Roman font, following conventional mathematical practice. Parentheses, digits, and operators are output in Roman font.

Spaces should be put around separate parts of the input. A common error is to type "f(pi)" without leaving spaces on both sides of the "pi". As a result, **eqn** does not recognize *pi* as a special word, and it appears as "f(pi)" in the output. A list of **eqn** names appears in Table 3.Z. Four-character **troff** names can also be used for anything **eqn** does not recognize, e.g., `\(pi` for the + sign.

The only way **eqn** can deduce that some sequence of letters may be special is if that sequence is separated from the letters on either side of it. This can be done by surrounding a special word by ordinary space, tab, or newline characters. Special words can also be made to stand out by surrounding them with tildes or circumflexes, e.g.:

$$x = 2 \pi \text{~int~} \sin(\text{~omega~t~}) \text{~dt}$$

is much the same as the previous example, except tildes separate words like *sin*, *omega*, etc., and also add an extra space per tilde.

$$x = 2 \pi \int \sin(\omega t) dt$$

#### 4.4 Subscripts and Superscripts

Subscripts and superscripts are introduced by the keywords "sub" and "sup".

$$x^2 + y_k$$

is produced by

$$x \text{ sup } 2 + y \text{ sub } k$$

The **eqn** program takes care of all size changes and vertical motions needed to make the hard copy look right. The words "sub" and "sup" must be surrounded by spaces. A space or tilde is used to mark the end of a subscript or superscript. Return to the original base line is automatic.



Multiple levels of subscripts or superscripts are allowed. Subscripted subscripts and superscripted superscripts such as:

$x_{i_{sub\ 1}}$

produces

$x_{i_1}$

A subscript and superscript on the same thing are printed one above the other if the subscript comes first. The construct "something sub something sup something" is recognized as a special case.

$x_{i\ sup\ 2}$

is

$x_i^2$

Other than this special case, "sub" and "sup" group to the right

$x\ sup\ y\ sub\ z$

generates

$x^{y_z}$

not

$x_z^y$

A common erroneous expression is of the form

$y = (x\ sup\ 2) + 1$

which causes

$y = (x^2) + 1$



instead of the intended

$$Y = (x2) + 1$$

The error is in omitting the space (<sp>) delimiting superscripts. The correct expression is

$$y = (x \text{ sup } 2) + 1$$

#### 4.5 Braces

Complicated expressions can be formed by using braces ({ }) to keep objects together in unambiguous groups. Braces indicate what goes over what or what terms are to be grouped before applying another mathematical function.

Normally, the end of a subscript or superscript is marked by a space (tab or tilde, etc.). If the subscript or superscript is something that has to be typed with spaces in it, braces are used to mark the beginning and end. The input

$$e \text{ sup } \{i \text{ omega } t\}$$

produces

$$e^{i\omega t}$$

Braces can be used to force **eqn** to treat something as a unit or just to make the intent perfectly clear.

Braces can occur within braces if necessary. The statement

$$e \text{ sup } \{i \text{ pi sup } \{\text{rho } + 1\}\}$$

generates

$$e^{i\pi^{\rho+1}}$$

A general rule is that an arbitrarily complicated string enclosed in braces can be used in place of a single character (such as  $x$ ). The **eqn** program administers formatting details. In all cases, the correct number of braces are to be used. Omitting one or adding an extra one causes **eqn** to complain.

The braces convention is an example of the power of using a recursive grammar to define the language. It is part of the language that dictates that if a construct can appear in some context, then any expression within braces can also occur in that context.

#### 4.6 Fractions

Fractions are specified with the keyword **over**.

$$a+b \text{ over } c+d+e = 1$$

produces



$$\frac{a+b}{c+d+e} = 1$$

The line is made the correct length and positioned automatically. When there is both an "over" and a "sup" in the same expression, **eqn** performs the "sup" first.

$$-b \text{ sup } 2 \text{ over } \pi$$

is

$$\frac{-b^2}{\pi}$$

#### 4.7 Square Roots

There is a *sqrt* operator for making square roots of the appropriate size.

$$x = \{-b \pm \text{sqrt}\{b \text{ sup } 2 - 4ac\} \text{ over } 2a$$

yields

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Since large radicals look poor on some typesetters, *sqrt* is not recommended for tall expressions.

#### 4.8 Summations, Integrals, and Similar Constructions

Summations, integrals, and similar constructions are easy.

$$\text{sum from } i=0 \text{ to } \{i= \text{inf}\} x \text{ sup } i$$

produces

$$\sum_{i=0}^{i=\infty} x^i$$

Braces indicate where the upper part *i= inf* begins and ends. None are necessary for the lower part *i=0* because it contains no spaces. Braces will never hurt; but if "from" and "to" parts contain any spaces, braces must be put around them.

The "from" and "to" parts are optional; but if both are used, they have to occur in that order.



Other useful characters can replace the *sum* in the above example. They are

int prod union inter

which become, respectively

$\int \prod \cup \cap$

Since characters before the "from" can be anything, even something in braces, "from-to" can often be used in unexpected ways.

$\lim_{n \rightarrow \infty} x_n = 0$

is

$\lim_{n \rightarrow \infty} x_n = 0$

#### 4.9 Size and Font Changes

Although *eqn* makes an attempt to use correct sizes and fonts, there are times when default assumptions are not what is wanted. Slides and transparencies often require larger characters than normal text. Thus size and font changing commands are also provided. By default, equations are set in 10-point type with standard mathematical conventions to determine what characters are in Roman and italic font. Size and font changes are made with *size n* and *Roman*, *italic*, *bold*, or *fat* operations. Like the "sub" and "sup" keywords, size and font changes affect only the string that follows and revert to the normal situation afterward. Thus:

**bold x y**

is

**xy**

Braces can be used if something more complicated than a single letter is to be affected.

Legal sizes which may follow *size* are

6, 7, 8, 9, 10, 11, 12, 14,  
16, 18, 20, 22, 24, 28, 36.

The size can also be changed by a given amount. For example:

*size +2*

makes the size two points larger. This has the advantage that knowledge of the current size is not necessary.

If fonts other than Roman, italic, and bold are to be used, the *font X* statement (*X* is a 1-character troff name or number for the font) can be used. Since *eqn* is tuned for Roman, italic, and bold, other fonts may not give as good an appearance.



The *fat* operation takes the current font and widens it by overstriking.

If an entire document is to be in a nonstandard size or font, it is a nuisance to write out a size and font change for each equation. Accordingly, a global size or font can be set that thereafter affects all equations. The following statements would appear at the beginning of any equation to set the size to 16 and the font to Roman:

```
.EQ
gsize 16
gfont R
...
.EN
```

In place of **R**, any of the **troff** font names may be used. The size after *gsize* can be a relative change with + or -.

Generally, *gsize* and *gfont* appear at the beginning of a document. They can also appear throughout a document. The global font and size can be changed as often as needed; for example, in a footnote in which the size of equations should match the size of the footnote text. Footnote text is usually two points smaller than the main text. Global size should be reset at the end of the footnote.

#### 4.10 Diacritical Marks

Diacritical marks, a problem in traditional typesetting, are straightforward. There are several words used to get funny marks on top of letters.

|          |                          |
|----------|--------------------------|
| x dot    | $\dot{x}$                |
| x dotdot | $\ddot{x}$               |
| x hat    | $\hat{x}$                |
| x tilde  | $\tilde{x}$              |
| x vec    | $\vec{x}$                |
| x dyad   | $\overleftrightarrow{x}$ |
| x bar    | $\bar{x}$                |
| x under  | $\underline{x}$          |

The diacritical mark is placed at the correct height, and *bar* and *under* are made the right length for the entire construction. Other marks are centered. An example of an expression using diacritical marks is:

$$\dot{x} + \hat{x} + \tilde{y} + \vec{X} + \ddot{Y} = \overline{z} + \underline{Z}$$



It is made by typing

$$\begin{array}{l} x \text{ dot under} + x \text{ hat} + y \text{ tilde} \\ + X \text{ hat} + Y \text{ dotdot} = z + Z \text{ bar} \end{array}$$

#### 4.11 Quoted Text

An input entirely within quotes ("...") is not subject to font changes or spacing adjustments normally done by the typesetting program. This provides for individual spacing and adjusting if needed. For example:

italic " sin(x) " + sin (x)

produces

$$\sin(x) + \sin(x)$$

Quotes are also used to get braces and other **eqn** keywords printed.

" { size alpha } "

prints

$$\{ \textit{size alpha} \}$$

and

roman " { size alpha } "

prints

$$\{ \text{size alpha} \}$$

The construction " " is often used as a place-holder when grammatically **eqn** needs something, but nothing is actually wanted on the output.

#### 4.12 Aligning Equations

Sometimes it is necessary to align a series of equations at a horizontal position, often at an equals sign. This is done with two operations called *mark* and *lineup*.

The word *mark* may appear once at any place in an equation. It remembers the horizontal position where it appeared. Successive equations can contain one occurrence of the word *lineup*. The place where *lineup* appears is made to line up with the place marked by the previous *mark* if at all possible. For example:

```
.EQ I
x+y mark = z
.EN
.EQ I
x lineup = 1
.EN
```

produces

$$\begin{array}{l} x+y = z \\ x = 1 \end{array}$$



When `eqn` and `-ms` are used, either `.EQ I` or `.EQ L` should be used. The `mark` and `lineup` operations do not work with centered equations. Also, `mark` does not look ahead.

`x mark =1`

...

`x+y lineup =z`

is not going to work because there is not room for the `x+y` part after the `mark` remembers where the `x` is.

#### 4.13 Big Brackets

Keywords "left" and "right" are used to make braces, brackets, parentheses, and vertical bars the correct height.

`left [ x+y over 2a right ] = 1`

produces

$$\left[ \frac{x+y}{2a} \right] = 1$$

To get large brackets [], braces {}, parentheses (), and bars # around information that exists on more than one line, the `left` and `right` commands are used.

`left { a over b + 1 right }`  
`= left ( c over d right )`  
`+ left [ e right ]`

is

$$\left\{ \frac{a}{b} + 1 \right\} = \left( \frac{c}{d} \right) + [e]$$

The resulting brackets are made large enough to cover whatever they enclose. Other characters can be used besides these, but they are not likely to look very good. One exception is the `floor` and `ceiling` characters.

`left floor x over y right floor`  
`<= left ceiling a over b right ceiling`

produces



$$\left| \frac{x}{y} \right| < \left| \frac{a}{b} \right|$$

Braces are larger than brackets and parentheses because they are made up of three, five, seven, etc., pieces while brackets can be made up of two, three, etc. Large left and right parentheses often look strange because the character set is poorly designed.

The *right* keyword may be omitted. A "left something" need not have a corresponding "right something". If the *right* part is omitted, braces are put around the thing that the left bracket is to encompass. Otherwise, resulting brackets may be too large. If the *left* part is to be omitted, things are more complicated because technically a *right* cannot exist without a corresponding *left*. Instead the following input will do:

left " " ..... right)

The left " " means a "left nothing", which satisfies the rules without hurting the output.

#### 4.14 Piles

Large brackets, etc., are often used with another facility, called *piles*, which make vertical piles of objects. Elements of the pile (there can be any number) are centered one above another, at the right height for most purposes. The keyword *above* is used to separate the pieces; braces are used around the entire list. Elements of a pile can be as complicated as needed, even containing more piles.

Three other forms of pile exist:

- *lpile* makes a pile with the elements left-justified
- *rpile* makes a right-justified pile
- *cpile* makes a centered pile, just like *pile*.

Vertical spacing between pieces is somewhat larger for *l-*, *r-*, and *cpiles* than it is for ordinary piles. For example, to get

$$\text{sign}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases}$$

the following is input:

```
sign (x) == left {
  rpile {1 above 0 above -1}
  ~lpile {if above if above if}
  ~lpile {x>0 above x=0 above x<0}
```

The *left {* construction makes a left brace large enough to enclose the *rpile {...}*, which is a right-justified pile of "above ... above ...". The *lpile* construction makes a left-justified pile.



## 4.15 Matrices

It is possible to make matrices. For example, to make a neat array like

$$\begin{array}{cc} x_1 & x^2 \\ y_1 & y^2 \end{array}$$

the following is the input:

```
matrix {
  ccol { x sub i above y sub i }
  ccol { x sup 2 above y sup 2 }
}
```

This produces a matrix with two centered columns. Elements of the columns are then listed just as for a pile, each element separated by the word "above". The *lcol* or *rcol* can also be used to left- or right-justify columns. Each column can be separately adjusted, and there can be as many columns as desired.

The reason for using a matrix instead of two adjacent piles is if the elements of the piles are not all the same height they will not line up properly. A matrix forces them to line up because it looks at the entire structure before deciding the spacing to use.

**Note:** Each column must have the same number of elements.

## 4.16 In-Line Equations

In a mathematical document, it is necessary to follow mathematical conventions in display equations and in text. Making variable names (such as *x*) italic is one instance. Although this could be done by surrounding the appropriate parts with **.EQ** and **.EN**, the continual repetition of **.EQ** and **.EN** is a nuisance. Furthermore, with **-mm**, **.EQ** and **.EN** imply a displayed equation.

The **eqn** program provides a shorthand notation for short in-line equations. Two characters can be defined to mark the left and right ends of a short in-line equation, and then expressions in the middle of text lines can be typed. If added to the beginning of the document, the three lines

```
.EQ
delim $$
.EN
```

set both the left and right delimiter characters to dollar signs. A sample input would be

Let  $\alpha_i$  be the primary variable, and let  $\beta$  be zero. Then it can be shown that  $x_{1i}$  is  $\geq 0$ .

This works as expected—space characters, newline characters, etc., are significant in the text, but not in the equation part itself. Multiple equations can occur in a single input line.

Space is left before and after a line that contains in-line expressions so that a tall expression will not interfere with surrounding lines.



To turn off the delimiters:

```
.EQ
delim off
.EN
```

The following should be observed when using the in-line equations format:

- Do not use braces, tildes, circumflexes, or double quotes as delimiters.
- In-line font changes must be closed before in-line equations are encountered.

#### 4.17 Defines

There is a definition facility, so a user can say

```
define name '...'
```

at any time in the document. Henceforth, any occurrence of the "name" in an expression will be expanded into whatever was inside the double quotes in its definition. This lets users tailor the language to their own specifications. It is possible to redefine keywords like *sup* or *over*. For example, if the sequence

```
x sub i sub 1 + y sub i sub 1
```

appears repeatedly throughout a paper; retyping time can be saved each time by defining it.

```
define xy 'x sub i sub 1 + y sub i sub 1'
```

Makes *xy* a shorthand for whatever characters occur between the single quotes in the definition. Any character can be used instead of the quote to mark the ends of the definition as long as it does not appear inside the definition.

The above expression can now be input as follows:

```
.EQ
f(x) = xy ...
.EN
```

Each occurrence of *xy* will expand into its definition. Spaces (or their equivalent) are to be left around the name when used. The **eqn** program will identify it as special.

Although definitions can use previous definitions, as in:

```
.EQ
define xi 'x sub i'
define xil 'xi sub 1'
.EN
```

it is erroneous to define something in terms of itself. For instance:

```
define X 'roman X'
```

Since *X* is now defined in terms of itself, problems will result. However, if the following expression is used, the quotes protect the second *X*, and everything works fine.

```
define X 'roman "X"'
```



The **eqn** keywords can be redefined. By making / mean *over* with the following statement:

```
define / 'over'
```

or by redefining *over* as / with:

```
define over '/'
```

the keyword is redefined.

If different things are needed to be printed on a terminal and on the typesetter, symbols may be defined differently in **neqn** and **eqn**. This can be done with *ndefine* and *tdefine*. A definition made with *ndefine* takes effect when running **neqn**. When *tdefine* is used, the definition only applies for **eqn**. Names defined with *define* apply to both **eqn** and **neqn**.

#### 4.18 Local Motions

Although **eqn** tries to position things correctly on the paper, it occasionally needs tuning to make the output just right. Small extra horizontal spaces can be obtained with tilde and circumflex. By using *back n* and *fwd n*, small amounts are moved horizontally, where *n* is how far to move in 1/100's of an em (an em is about the width of the letter "m"). Thus, *back 50* moves back about half the width of an "m". Similarly, things can be moved up or down with an *up n* and a *down n*. As with *sub* or *sup*, local motions affect the next thing in the input, and this can be something arbitrarily complicated if it is enclosed in braces.

#### 4.19 Precedence

Precedence rules resolve the ambiguity in a construction like:

```
a sup 2 over b
```

The "sup" is defined to have a higher precedence than "over". A user can force a particular analysis by placing braces around expressions. If braces are not used to group functions, **eqn** will do operations in the following order:

```
dyad vec under bar tilde hat dot dotdot
fwd back down up
fat roman italic bold size
sub sup sqrt over
from to
```

The following operations group to the left:

```
over sqrt left right
```

All others group to the right.

#### 5. Troubleshooting

If a mistake is made in an equation, such as omitting a brace, having one too many braces, or having a "sup" with nothing before it, **eqn** produces the following message:

```
syntax error between lines x and y, file z
```

where *x* and *y* are approximately the lines between which the trouble occurred, and *z* is the name of the file in question. There are also self-explanatory messages that arise when a quote is omitted or **eqn** is run on a non-existent file. To check a document before printing



`eqn files >/dev/null`

discards the output but prints the message.

It is easy to leave out a dollar sign when used as delimiters. The `checkeq` program checks for misplaced or missing dollar signs and similar troubles.

In-line equations can be only so big because of an internal buffer in the `troff` formatter. If a "word overflow" message is received, the limit has been exceeded. Printing the equation as a displayed equation usually causes the message to go away. The "line overflow" message indicates that an even bigger buffer has been exceeded. In this case, the equation must be broken into two separate ones, marking each with a `.EQ ... .EN` delimiter. The `eqn` program does not warn about equations that are too long for one line.



The following font examples are printed in 12-point, with a vertical spacing of 14-point, and with non-alphanumeric characters separated by ¼ em space. The original Special Mathematical Font was prepared for Bell Laboratories by Wang Laboratories, Inc., of Hudson, New Hampshire. The Times Roman, Italic, and Bold are among the many standard fonts available.

# Times Roman

abcdefghijklmnopqrstuvwxyz  
 ABCDEFGHIJKLMNOPQRSTUVWXYZ  
 1234567890  
 ! \$ % & ( ) ' \* + - . , / : ; = ? [ ] |  
 • □ — — ¼ ½ ¾ fi fl ff ffi mfi ° † ' ¢ ® ©

*Times Italic*

abcdefghijklmnopqrstuvwxyz  
ABCDEFGHIJKLMNOPQRSTUVWXYZ  
1234567890  
!\$%&'()\*+,-./:;=?[ ]!  
• □ − − ¼ ½ ¾ fi fl ff m m ° † ' ¢ ® ©

## Times Bold

abcdefghijklmnopqrstuvwxyz  
ABCDEFGHIJKLMNOPQRSTUVWXYZ  
1234567890  
!\$%&'()\*+,-./:;=?[\\]  
•◻— - ¼ ½ ¾ fi fl ff m m ° † ' ¢ ®

## Special Mathematical Font

" '\ ^ \_ \$ % ' / < > { } # @ + - = \*  
α β γ δ ε ζ η θ ι κ λ μ ν ξ ο π ρ σ τ υ φ χ ψ ω  
Γ Δ Θ Λ Ξ Π Σ Υ Φ Ψ Ω  
√ ∓ ≥ ≤ ≡ ∼ ≈ ≠ → ← ↑ ↓ × ÷ ± ∪ ∩ ⊂ ⊆  
≧ ≯ ∂  
§ ∇ ∫ ∞ ∅ ∈ † ‡ § ¶ © ¤ ∘ ∙ √ ∣ ∩ ∪ ∖ ∗

**Fig. 3.1 — Font Style Examples**



Automatic sequence numbering of output lines may be requested with  
3 `.nm`. When in effect, a 3-digit, arabic number plus a digit-space is  
6 prepended to output text lines. Text lines are offset by four digit-spaces  
9 and otherwise retain their line length. A reduction in line length may  
12 be desired to keep the right margin aligned with an earlier margin.  
Blank lines, other vertical spaces, and lines generated by `.tl` are not  
numbered. Numbering can be temporarily suspended with `.nn` or with a  
`.nm` followed by a later `.nm +0`. In addition, a line number indent *I* and  
the number-text separation *S* may be specified in digit-spaces. Further,  
it can be specified that only those line numbers that are multiples of  
some number *M* are to be printed (the others will appear as blank  
number fields). Table 3.S is a summary and explanation of output line  
numbering requests.

As an example of output line numbering, paragraph portions of this  
15 figure are numbered with *M*=3: `.nm 1 3` was placed at the beginning;  
`.nm` was placed at the end of the first paragraph; and `.nm +0` was  
placed in front of this paragraph; and `.nm` placed at the end. Line  
18 lengths were also changed (by `\w'0000'u`) to keep the right side  
aligned. Another example is `.nm +5 5 x 3`, which turns on numbering  
with the line number of the next line to be five greater than the last  
21 numbered line, with *M*=5, spacing *S* untouched, and the indent *I* set to 3.

Fig. 3.2 — Example of Output Line Numbering



## INPUT:

```
.TS
box;
ccc
lll.
LanguageⓈ AuthorsⓈ Runs on
.sp
FortranⓈ ManyⓈ Almost anything
PL/1Ⓢ IBMⓈ 360/370
CⓈ BTLⓈ 11/45,H6000,370
BLISSⓈ Carnegie-MellonⓈ PDP-10,11
IDSⓈ HoneywellⓈ H6000
PascalⓈ StanfordⓈ 370
.TE
```

## OUTPUT:

| Language | Authors         | Runs on         |
|----------|-----------------|-----------------|
| Fortran  | Many            | Almost anything |
| PL/1     | IBM             | 360/370         |
| C        | BTL             | 11/45,H6000,370 |
| BLISS    | Carnegie-Mellon | PDP-10,11       |
| IDS      | Honeywell       | H6000           |
| Pascal   | Stanford        | 370             |

Fig. 3.3 — Table Using "box" Option



## INPUT:

```
.TS
allbox;
c s s
c c c
n n n.
AT&T Common Stock
Year① Price① Dividend
1971① 41-54① $2.60
2① 41-54① 2.70
3① 46-55① 2.87
4① 40-53① 3.24
5① 45-52① 3.40
6① 51-59① .95*
.TE
* (first quarter only)
```

## OUTPUT:

| AT&T Common Stock |       |          |
|-------------------|-------|----------|
| Year              | Price | Dividend |
| 1971              | 41-54 | \$2.60   |
| 2                 | 41-54 | 2.70     |
| 3                 | 46-55 | 2.87     |
| 4                 | 40-53 | 3.24     |
| 5                 | 45-52 | 3.40     |
| 6                 | 51-59 | .95*     |

\* (first quarter only)

Fig. 3.4 — Table Using "allbox" Option



## INPUT:

.TS  
 box;  
 c s s  
 cl cl c  
 ll ll n.  
 Major New York Bridges

— BridgeⓈ DesignerⓈ Length

— BrooklynⓈ J. A. RoeblingⓈ 1595  
 ManhattanⓈ G. LindenthalⓈ 1470  
 WilliamsburgⓈ L. L. BuckⓈ 1600

— QueensboroughⓈ Palmer &Ⓢ 1182  
 Ⓢ Hornbostel

—  
 ⓈⓈ 1380  
 TriboroughⓈ O. H. AmmannⓈ  
 ⓈⓈ 383

— Bronx WhitestoneⓈ O. H. AmmannⓈ 2300  
 Throgs NeckⓈ O. H. AmmannⓈ 1800  
 .TE

## OUTPUT:

| Major New York Bridges |                        |        |
|------------------------|------------------------|--------|
| Bridge                 | Designer               | Length |
| Brooklyn               | J. A. Roebling         | 1595   |
| Manhattan              | G. Lindenthal          | 1470   |
| Williamsburg           | L. L. Buck             | 1600   |
| Queensborough          | Palmer &<br>Hornbostel | 1182   |
| Triborough             | O. H. Ammann           | 1380   |
|                        |                        | 383    |
| Bronx Whitestone       | O. H. Ammann           | 2300   |
| Throgs Neck            | O. H. Ammann           | 1800   |

Fig. 3.5 — Table Using "Vertical bar" Key Letter Feature



## INPUT:

.TS  
 box;  
 L L L  
 L L \_  
 L L | LB  
 L L \_  
 L L L.  
 january① february① march  
 april① may  
 june① july① Months  
 august① september  
 october① november① december  
 .TE

## OUTPUT:

|         |           |          |
|---------|-----------|----------|
| january | february  | march    |
| april   | may       |          |
| june    | july      | Months   |
| august  | september |          |
| october | november  | december |

Fig. 3.6 — Table Using Horizontal Lines in Place of Key Letters



## INPUT:

```
.TS
box;
cfB s s s.
Composition of Foods

.T&
c l c s s
c l c s s
c l c l c l c.
Food⓪ Percent by Weight
\⓪_
\⓪ Protein⓪ Fat⓪ Carbo-
\⓪\⓪\⓪ hydrate

.T&
l l n l n l n.
Apples⓪ .4⓪ .5⓪ 13.0
Halibut⓪ 18.4⓪ 5.2⓪ ...
Lima beans⓪ 7.5⓪ .8⓪ 22.0
Milk⓪ 3.3⓪ 4.0⓪ 5.0
Mushrooms⓪ 3.5⓪ .4⓪ 6.0
Rye bread⓪ 9.0⓪ .6⓪ 52.7
.TE
```

## OUTPUT:

| Composition of Foods |                   |     |                   |
|----------------------|-------------------|-----|-------------------|
| Food                 | Percent by Weight |     |                   |
|                      | Protein           | Fat | Carbo-<br>hydrate |
| Apples               | .4                | .5  | 13.0              |
| Halibut              | 18.4              | 5.2 | ...               |
| Lima beans           | 7.5               | .8  | 22.0              |
| Milk                 | 3.3               | 4.0 | 5.0               |
| Mushrooms            | 3.5               | .4  | 6.0               |
| Rye bread            | 9.0               | .6  | 52.7              |

Fig. 3.7 — Table Using Additional Command Lines



## INPUT:

```
.TS
allbox;
cf l s s
c cw(2i) cw(2i)
l l l.
New York Area Rocks
.sp
EraⓈ FormationⓈ Age (years)
PrecambrianⓈ Reading ProngⓈ >1 billion
PaleozoicⓈ Manhattan ProngⓈ 400 million
MesozoicⓈ T{
.na
Newark Basin, incl.
Stockton, Lockatong, and Brunswick
formations
.ad
T}Ⓢ 200 million
CenozoicⓈ Coastal PlainⓈ T{
.na
On Long Island 30,000 years;
Cretaceous sediments redeposited
by recent glaciation
.ad
T}
.TE
```

## OUTPUT:

| New York Area Rocks |                                                                         |                                                                                             |
|---------------------|-------------------------------------------------------------------------|---------------------------------------------------------------------------------------------|
| Era                 | Formation                                                               | Age (years)                                                                                 |
| Precambrian         | Reading Prong                                                           | >1 billion                                                                                  |
| Paleozoic           | Manhattan Prong                                                         | 400 million                                                                                 |
| Mesozoic            | Newark Basin, incl. Stockton,<br>Lockatong, and Brunswick<br>formations | 200 million                                                                                 |
| Cenozoic            | Coastal Plain                                                           | On Long Island 30,000 years;<br>Cretaceous sediments<br>redeposited by recent<br>glaciation |

Fig. 3.8 — Table Using Text Blocks



TABLE 3.A

CROSS REFERENCE  
REQUEST NAME TO TABLE NUMBER

| REQUEST<br>NAME | TABLE<br>NUMBER | REQUEST<br>NAME | TABLE<br>NUMBER | REQUEST<br>NAME | TABLE<br>NUMBER | REQUEST<br>NAME | TABLE<br>NUMBER |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| ab              | 3.Y             | el              | 3.T             | ls              | 3.H             | rd              | 3.V             |
| ad              | 3.G             | em              | 3.J             | lt              | 3.R             | rm              | 3.J             |
| af              | 3.M             | eo              | 3.O             | mc              | 3.X             | rn              | 3.J             |
| am              | 3.J             | ev              | 3.U             | mk              | 3.F             | rr              | 3.M             |
| as              | 3.J             | ex              | 3.V             | na              | 3.G             | rs              | 3.H             |
| bd              | 3.C             | fc              | 3.N             | ne              | 3.F             | rt              | 3.F             |
| bp              | 3.F             | fi              | 3.G             | nf              | 3.G             | so              | 3.W             |
| br              | 3.G             | fl              | 3.X             | nh              | 3.Q             | sp              | 3.H             |
| c2              | 3.O             | fp              | 3.C             | nm              | 3.S             | ss              | 3.E             |
| cc              | 3.O             | ft              | 3.C             | nn              | 3.S             | sv              | 3.H             |
| ce              | 3.G             | hc              | 3.Q             | nr              | 3.M             | ta              | 3.N             |
| ch              | 3.J             | hw              | 3.Q             | ns              | 3.H             | tc              | 3.N             |
| co              | 3.X             | hy              | 3.Q             | nx              | 3.W             | ti              | 3.I             |
| cs              | 3.E             | ie              | 3.T             | os              | 3.H             | tl              | 3.R             |
| cu              | 3.O             | if              | 3.T             | pc              | 3.R             | tm              | 3.X             |
| da              | 3.J             | ig              | 3.X             | pi              | 3.W             | tr              | 3.O             |
| de              | 3.J             | in              | 3.I             | pl              | 3.F             | uf              | 3.O             |
| di              | 3.J             | it              | 3.J             | pm              | 3.X             | ul              | 3.O             |
| ds              | 3.J             | lc              | 3.N             | pn              | 3.F             | vs              | 3.H             |
| dt              | 3.J             | lg              | 3.O             | po              | 3.F             | wh              | 3.J             |
| ec              | 3.O             | ll              | 3.I             | ps              | 3.E             |                 |                 |



TABLE 3.8

ESCAPE SEQUENCES FOR  
CHARACTERS, INDICATORS, AND FUNCTIONS

| ESCAPE SEQUENCE   | MEANING                                                                                                   |
|-------------------|-----------------------------------------------------------------------------------------------------------|
| \ (Note)          | \ (to prevent or delay the interpretation of \ )                                                          |
| \ '               | Acute accent; equivalent to \ (aa                                                                         |
| \ `               | Grave accent; equivalent to \ (ga                                                                         |
| \ -               | Minus sign in the current font                                                                            |
| \ .(Note)         | Period (dot) (see de)                                                                                     |
| \ <space>         | Unpaddable space-size space character                                                                     |
| \ 0               | Digit width space                                                                                         |
| \ !               | 1/6 em narrow space character (zero width in the nroff formatter)                                         |
| \ ^               | 1/12 em half-narrow space character (zero width in the nroff formatter)                                   |
| \ &               | Nonprinting zero width character                                                                          |
| \ !               | Transparent line indicator                                                                                |
| \ "(Note)         | Beginning of comment                                                                                      |
| \ \$N             | Interpolate argument (1 < N > 9)                                                                          |
| \ %               | Default optional hyphenation character                                                                    |
| \ (xx             | Character named xx                                                                                        |
| \ *x,\*(xx(Note)  | Interpolate string x or xx                                                                                |
| \ {               | Begin conditional input                                                                                   |
| \ }               | End conditional input                                                                                     |
| \ <newline>(Note) | Concealed (ignored) newline character                                                                     |
| \ a(Note)         | Noninterpreted leader character                                                                           |
| \ b'abc..'        | Bracket building function                                                                                 |
| \ c               | Interrupt text processing                                                                                 |
| \ d               | Forward (down) 1/2 em vertical motion (1/2 line in the nroff formatter)                                   |
| \ e               | Printable version of current escape character                                                             |
| \ fx,\f(xx,\fN    | Change to font named x or xx or position N                                                                |
| \ gx,\g(xx        | Return the .af-type format of the register x or xx. Returns nothing if x (xx) has not yet been referenced |
| \ h'N             | Local horizontal motion; move right N (negative left)                                                     |
| \ jx,\j(xx        | Mark the current horizontal output position in register x or xx                                           |
| \ kx              | Mark horizontal input place in register x                                                                 |
| \ l'Nc'           | Horizontal line drawing function (optionally with c)                                                      |
| \ L'Nc'           | Vertical line drawing function (optionally with c)                                                        |
| \ nx,\n(xx(Note)  | Interpolate number register x or xx                                                                       |
| \ o'abc..'        | Overstrike characters a, b, c, ...                                                                        |
| \ p               | Break and spread output line                                                                              |
| \ r               | Reverse 1 em vertical motion (reverse line in the nroff formatter)                                        |
| \ sN,\s±N         | Point-size change function                                                                                |
| \ t(Note)         | Noninterpreted horizontal tab                                                                             |
| \ u               | Reverse (up) 1/2 em vertical motion (1/2 line in the nroff formatter)                                     |
| \ v'N             | Local vertical motion; move down N (negative up)                                                          |
| \ w'string'       | Interpolate width of string                                                                               |
| \ x'N             | Extra line-space function (negative before, positive after)                                               |
| \ zc              | Print c with zero width (without spacing)                                                                 |
| \ X               | Any character not listed above                                                                            |

**Note:** Interpreted in copy mode.



TABLE 3.C

## FONT CONTROL REQUESTS

| REQUEST FORM     | INITIAL VALUE | IF NO ARGUMENT | EXPLANATION                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|------------------|---------------|----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| .bd <i>F N</i>   | off           | —              | Embolden font <i>F</i> by <i>N</i> -1 units. Characters in font <i>F</i> will be artificially emboldened by printing each one twice, separated by <i>N</i> -1 basic units. A reasonable value for <i>N</i> is 3 when the character size is in the vicinity of 10 points. If <i>N</i> is missing, the embolden mode is turned off. The mode must still (or again) be in effect when the characters are physically printed. There is no effect in the <b>nroff</b> formatter. |
| .bd <i>S F N</i> | off           | —              | Embolden special font when current font is <i>F</i> . The characters in the special font will be emboldened whenever the current font is <i>F</i> . The mode must still (or again) be in effect when the characters are physically printed. There is no effect in the <b>nroff</b> formatter.                                                                                                                                                                               |
| .fp <i>N F</i>   | R,I,B,S       | ignored        | Font position. A font named <i>F</i> is mounted on position <i>N</i> (1 through 4). It is a fatal error if <i>F</i> is not known. The phototypesetter has four fonts physically mounted. Each font consists of a film strip which can be mounted on a numbered quadrant of a wheel. The default mounting sequence assumed by the <b>troff</b> formatter is R, I, B, and S on positions 1, 2, 3, and 4. <i>2-letter name: .fp 2 fx</i>                                       |
| .ft <i>F</i>     | Roman         | previous       | Change to font <i>F</i> , where <i>F</i> is <b>x</b> , <b>xx</b> , 1 through 4, or <b>P</b> . Font <b>P</b> means the previous font. For font changes within a line of text, sequences <b>\fx</b> , <b>\f(xx</b> , or <b>\fN</b> can be used. Relevant parameters are a part of the current environment.                                                                                                                                                                    |



TABLE 3.D

## NAMING CONVENTION FOR NON-ASCII CHARACTERS

Non-ASCII characters and *minus* on the standard fonts.

| Char | Input Name | Character Name     | Char | Input Name | Character Name |
|------|------------|--------------------|------|------------|----------------|
| '    |            | close quote        | fi   | \(fi       | fi             |
| '    |            | open quote         | fl   | \(fl       | fl             |
| —    | \(em       | 3/4 Em dash        | ff   | \(ff       | ff             |
| -    |            | hyphen or          | ffi  | \(Fi       | ffi            |
| -    | \(hy       | hyphen             | ffl  | \(Fl       | ffl            |
| -    | \(—        | current font minus | °    | \(de       | degree         |
| •    | \(bu       | bullet             | †    | \(dg       | dagger         |
| □    | \(sq       | square             | '    | \(fm       | foot mark      |
| —    | \(ru       | rule               | ¢    | \(ct       | cent sign      |
| ¼    | \(14       | 1/4                | ®    | \(rg       | registered     |
| ½    | \(12       | 1/2                | ©    | \(co       | copyright      |
| ¾    | \(34       | 3/4                |      |            |                |

Non-ASCII characters and ` , ` \_ , + , - , = , and \* on the special font.

| Char | Input Name | Character Name             | Char | Input Name | Character Name |
|------|------------|----------------------------|------|------------|----------------|
| +    | \(pl       | math plus                  | κ    | \(*k       | kappa          |
| -    | \(mi       | math minus                 | λ    | \(*l       | lambda         |
| =    | \(eq       | math equals                | μ    | \(*m       | mu             |
| *    | \(*        | math star                  | ν    | \(*n       | nu             |
| §    | \(sc       | section                    | ξ    | \(*c       | xi             |
| ·    | \(aa       | acute accent               | ο    | \(*o       | omicron        |
| ·    | \(ga       | grave accent               | π    | \(*p       | pi             |
| _    | \(ul       | underrule                  | ρ    | \(*r       | rho            |
| /    | \(sl       | slash (matching backslash) | σ    | \(*s       | sigma          |
| α    | \(*a       | alpha                      | τ    | \(ts       | terminal sigma |
| β    | \(*b       | beta                       | τ    | \(*t       | tau            |
| γ    | \(*g       | gamma                      | υ    | \(*u       | upsilon        |
| δ    | \(*d       | delta                      | φ    | \(*f       | phi            |
| ε    | \(*e       | epsilon                    | χ    | \(*x       | chi            |
| ζ    | \(*z       | zeta                       | ψ    | \(*q       | psi            |
| η    | \(*y       | eta                        | ω    | \(*w       | omega          |
| θ    | \(*h       | theta                      | Λ    | \(*A       | Alpha†         |
| ι    | \(*i       | iota                       | Β    | \(*B       | Beta†          |

†Mapped into uppercase English letters in the font mounted on font position one.



TABLE 3.D (Contd)  
NAMING CONVENTION FOR NON-ASCII CHARACTERS

| Char              | Input Name | Character Name    | Char                      | Input Name | Character Name                                 |
|-------------------|------------|-------------------|---------------------------|------------|------------------------------------------------|
| $\Gamma$          | \(*G       | Gamma             | $\div$                    | \(di       | divide                                         |
| $\Delta$          | \(*D       | Delta             | $\pm$                     | \(+-       | plus-minus                                     |
| $\epsilon$        | \(*E       | Epsilon†          | $\cup$                    | \(cu       | cup (union)                                    |
| $\zeta$           | \(*Z       | Zeta†             | $\cap$                    | \(ca       | cap (intersection)                             |
| $\eta$            | \(*Y       | Eta†              | $\subset$                 | \(sb       | subset of                                      |
| $\theta$          | \(*H       | Theta             | $\supset$                 | \(sp       | superset of                                    |
| $\iota$           | \(*I       | Iota†             | $\imath$                  | \(ib       | improper subset                                |
| $\kappa$          | \(*K       | Kappa†            | $\Im$                     | \(ip       | improper superset                              |
| $\lambda$         | \(*L       | Lambda            | $\infty$                  | \(if       | infinity                                       |
| $\mu$             | \(*M       | Mu†               | $\partial$                | \(pd       | partial derivative                             |
| $\nu$             | \(*N       | Nu†               | $\nabla$                  | \(gr       | gradient                                       |
| $\xi$             | \(*C       | Xi                | $\neg$                    | \(no       | not                                            |
| $\omicron$        | \(*O       | Omicron†          | $\int$                    | \(is       | integral sign                                  |
| $\pi$             | \(*P       | Pi                | $\propto$                 | \(pt       | proportional to                                |
| $\rho$            | \(*R       | Rho†              | $\emptyset$               | \(es       | empty set                                      |
| $\sigma$          | \(*S       | Sigma             | $\in$                     | \(mo       | member of                                      |
| $\tau$            | \(*T       | Tau†              | $ $                       | \(br       | box vertical rule                              |
| $\upsilon$        | \(*U       | Upsilon           | $\ddagger$                | \(dd       | double dagger                                  |
| $\phi$            | \(*F       | Phi               | $\text{right hand}$       | \(rh       | right hand                                     |
| $\chi$            | \(*X       | Chi†              | $\text{left hand}$        | \(lh       | left hand                                      |
| $\psi$            | \(*Q       | Psi               | $\text{Bell System logo}$ | \(bs       | Bell System logo                               |
| $\omega$          | \(*W       | Omega             | $ $                       | \(or       | or                                             |
| $\sqrt{\quad}$    | \(sr       | square root       | $\bigcirc$                | \(ci       | circle                                         |
| $\sqrt[n]{\quad}$ | \(rn       | root en extender  | $\{$                      | \(lt       | left top of big curly bracket                  |
| $\geq$            | \(>=       | $\geq$            | $\{$                      | \(lb       | left bottom                                    |
| $\leq$            | \(<=       | $\leq$            | $\}$                      | \(rt       | right top                                      |
| $\equiv$          | \(==       | identically equal | $\}$                      | \(rb       | right bot                                      |
| $\approx$         | \(=        | approx =          | $\{$                      | \(lk       | left center of big curly bracket               |
| $\sim$            | \(ap       | approximates      | $\}$                      | \(rk       | right center of big curly bracket              |
| $\neq$            | \(!=       | not equal         | $ $                       | \(bv       | bold vertical                                  |
| $\rightarrow$     | \(->       | right arrow       | $\lfloor$                 | \(lf       | left floor (left bottom of big square bracket) |
| $\leftarrow$      | \(<-       | left arrow        | $\rfloor$                 | \(rf       | right floor (right bottom)                     |
| $\uparrow$        | \(ua       | up arrow          | $\lceil$                  | \(lc       | left ceiling (left top)                        |
| $\downarrow$      | \(da       | down arrow        | $\rceil$                  | \(rc       | right ceiling (right top)                      |
| $\times$          | \(mu       | multiply          |                           |            |                                                |

†Mapped into uppercase English letters in the font mounted on font position one.



TABLE 3.E

## CHARACTER SIZE CONTROL REQUESTS

| REQUEST FORM     | INITIAL VALUE | IF NO ARGUMENT | EXPLANATION                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|------------------|---------------|----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| .cs <i>F N M</i> | off           | —              | Set constant character space (width) mode on for font <i>F</i> (if mounted). The width of every character is assumed to be $N/36$ ems. If <i>M</i> is absent, the em is that of the character point size; if <i>M</i> is given, the em is <i>M</i> points. All affected characters are centered in this space including those with an actual width larger than this space. Special font characters occurring while the current font is <i>F</i> are also so treated. If <i>N</i> is absent, the mode is turned off. The mode must still (or again) be in effect when the characters are printed. There is no effect in the <b>nroff</b> formatter. |
| .ps $\pm N$      | 10 point      | previous       | Set point size to $\pm N$ . Any positive size value may be requested; if invalid, the next larger valid size will result (maximum of 36). A paired sequence $+N$ , $-N$ will work because the previous requested value is remembered. For point size changes within a line of text, sequences $\backslash sN$ or $\backslash s\pm N$ can be used. Relevant parameters are a part of the current environment. There is no effect in the <b>nroff</b> formatter.                                                                                                                                                                                     |
| .ss <i>N</i>     | 12/36 em      | ignored        | Set space-character size to $N/36$ ems. This size is the minimum word spacing in adjusted text. Relevant parameters are a part of the current environment. There is no effect in the <b>nroff</b> formatter.                                                                                                                                                                                                                                                                                                                                                                                                                                       |



TABLE 3.F  
PAGE CONTROL REQUESTS

| REQUEST FORM | INITIAL VALUE* | IF NO ARGUMENT | EXPLANATION                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|--------------|----------------|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| .bp $\pm N$  | $N=1$          | —              | Begin page. The current page is ejected and a new page is begin. If $\pm N$ is given, the new page number will be $\pm N$ . The page number indicator ( $N$ ) is ignored if not specified in the request. The request causes a break. The use of “ ” as the control character (instead of “.”) suppresses the break function. The request with no $N$ is inhibited by the .ns request.                                                                                                                                                                                                                                                                                                                                                                                                                      |
| .mk $R$      | none           | internal       | Mark current vertical place in an internal register (associated with the current diversion level) or in register $R$ , if given. The request is used in conjunction with “return to marked vertical place in current diversion” request (.rt). Mode or relevant parameters are associated with current diversion level.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| .ne $N$      | —              | $N=1 V$        | <p>Need <math>N</math> vertical spaces. The vertical space indicator (<math>N</math>) is ignored if not specified in the request.</p> <ul style="list-style-type: none"> <li>• If the distance to the next trap position (<math>D</math>) is less than <math>N</math>, a forward vertical space of size <math>D</math> occurs which will spring the trap.</li> <li>• If there are no remaining traps on the page, <math>D</math> is the distance to the bottom of the page.</li> <li>• If <math>D</math> is less than vertical spacing (<math>V</math>), another line could still be output and spring the trap.</li> </ul> <p>In a diversion, <math>D</math> is the distance to the diversion trap (if any) or is very large. Mode or relevant parameters are associated with current diversion level.</p> |
| .pl $\pm N$  | 11in           | 11in           | Page length set to $\pm N$ . The internal limitation is about 75 inches in the troff formatter and 136 inches in the nroff formatter. Current page length is available in the .p register. The page length indicator ( $N$ ) is ignored if not specified in the request.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| .pn $\pm N$  | $N=1$          | ignored        | Page number. The next page (when it occurs) will have the page number $\pm N$ . The request must occur before the initial pseudopage transition to affect the page number of the first page. The current page number is in the % register.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

\*Values separated by “;” are for the nroff and troff formatters, respectively.



TABLE 3.F (Contd)

## PAGE CONTROL REQUESTS

| REQUEST FORM | INITIAL VALUE* | IF NO ARGUMENT | EXPLANATION                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|--------------|----------------|----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| .po $\pm N$  | 0;<br>26/27in  | previous       | Page offset. The current left margin is set to $\pm N$ . The page offset indicator ( $N$ ) is ignored if not specified in the request. The <b>troff</b> formatter initial value provides about 1 inch of paper margin including the physical typesetter margin of 1/27 inch. In the <b>troff</b> formatter, the maximum (line-length) + (page-offset) is about 7.54 inches. The current page offset is available in the .o register.                                                                   |
| .rt $\pm N$  | none           | internal       | Return (upward only) to marked vertical place in current diversion. If $\pm N$ (with respect to place) is given, the vertical place is $\pm N$ from the top of the page or diversion. If $N$ is absent, the vertical place is marked by a previous .mk. The .sp request may be used in all cases instead of .rt by spacing to the absolute place stored in an explicit register; e.g., using the sequence .mk R ... .sp/\nRu. Mode or relevant parameters are associated with current diversion level. |

\*Values separated by ";" are for the **nroff** and **troff** formatters, respectively.



TABLE 3.G

## TEXT FILLING, ADJUSTING, AND CENTERING REQUESTS

| REQUEST FORM | INITIAL VALUE            | IF NO ARGUMENT | EXPLANATION                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |          |                 |   |                         |   |                          |   |        |        |                     |        |           |
|--------------|--------------------------|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------|-----------------|---|-------------------------|---|--------------------------|---|--------|--------|---------------------|--------|-----------|
| .ad <i>N</i> | adjust                   | adjust         | <p>Adjust. Output lines are adjusted with mode <i>N</i>. If the type indicator (<i>N</i>) is present, the adjustment type is as follows:</p> <table><tr><th><i>N</i></th><th>ADJUSTMENT TYPE</th></tr><tr><td>l</td><td>adjust left margin only</td></tr><tr><td>r</td><td>adjust right margin only</td></tr><tr><td>c</td><td>center</td></tr><tr><td>b or n</td><td>adjust both margins</td></tr><tr><td>absent</td><td>unchanged</td></tr></table> <p>The adjustment type indicator <i>N</i> may also be a number obtained from the .j register. If fill mode is not on, adjustment will be deferred. Relevant parameters are a part of the current environment.</p> | <i>N</i> | ADJUSTMENT TYPE | l | adjust left margin only | r | adjust right margin only | c | center | b or n | adjust both margins | absent | unchanged |
| <i>N</i>     | ADJUSTMENT TYPE          |                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |          |                 |   |                         |   |                          |   |        |        |                     |        |           |
| l            | adjust left margin only  |                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |          |                 |   |                         |   |                          |   |        |        |                     |        |           |
| r            | adjust right margin only |                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |          |                 |   |                         |   |                          |   |        |        |                     |        |           |
| c            | center                   |                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |          |                 |   |                         |   |                          |   |        |        |                     |        |           |
| b or n       | adjust both margins      |                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |          |                 |   |                         |   |                          |   |        |        |                     |        |           |
| absent       | unchanged                |                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |          |                 |   |                         |   |                          |   |        |        |                     |        |           |
| .br          | —                        | —              | Break. Filling of the line currently being collected is stopped, and the line is output without adjustment. Text lines beginning with space characters and empty text lines (blank lines) also cause a break.                                                                                                                                                                                                                                                                                                                                                                                                                                                           |          |                 |   |                         |   |                          |   |        |        |                     |        |           |
| .ce <i>N</i> | off                      | <i>N</i> =1    | Center. The next <i>N</i> input text lines are centered within the current line-length. If <i>N</i> =0, any residual count is cleared. A break occurs after each of the <i>N</i> input lines. If the input line is too long, it will be left adjusted. The request normally causes a break. Relevant parameters are a part of the current environment.                                                                                                                                                                                                                                                                                                                  |          |                 |   |                         |   |                          |   |        |        |                     |        |           |
| .fi          | fill                     | —              | Fill mode. The request causes a break. Subsequent output lines are filled to provide an even right margin. Relevant parameters are a part of the current environment.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |          |                 |   |                         |   |                          |   |        |        |                     |        |           |
| .na          | adjust                   | —              | No adjust. No output line adjusting is done. Since adjustment is turned off, the right margin will be ragged. Adjustment type for the .ad request is not changed. Output line filling still occurs if fill mode is on. Relevant parameters are a part of the current environment.                                                                                                                                                                                                                                                                                                                                                                                       |          |                 |   |                         |   |                          |   |        |        |                     |        |           |
| .nf          | fill                     | —              | No-fill mode. Subsequent output lines are neither filled nor adjusted. The request normally causes a break. Input text lines are copied directly to output lines without regard for the current line length. Relevant parameters are a part of the current environment.                                                                                                                                                                                                                                                                                                                                                                                                 |          |                 |   |                         |   |                          |   |        |        |                     |        |           |



TABLE 3.H

## VERTICAL SPACING REQUESTS

| REQUEST FORM | INITIAL VALUE*  | IF NO ARGUMENT       | EXPLANATION                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|--------------|-----------------|----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| .ls <i>N</i> | <i>N</i> =1     | previous             | Line spacing set to $\pm N$ . Output <i>N</i> -1 blank lines ( <i>V</i> <sub>3</sub> ) after each output text line. If the text or previous appended blank line reached a trap position, appended blank lines are omitted. Relevant parameters are a part of the current environment.                                                                                                                                                                                                    |
| .ns          | space           | —                    | Set no-space mode on. The no-space mode inhibits .sp and .bp requests without a next page number. It is turned off when a line of output occurs or with the .rs request. Mode or relevant parameters are associated with current diversion level.                                                                                                                                                                                                                                        |
| .os          | —               | —                    | Output saved vertical space. The request is used to output a block of vertical space requested by an earlier .sv request. The no-space mode has no effect.                                                                                                                                                                                                                                                                                                                               |
| .rs          | —               | —                    | Restore spacing. The no-space mode is turned off. Mode or relevant parameters are associated with current diversion level.                                                                                                                                                                                                                                                                                                                                                               |
| .sp <i>N</i> | —               | <i>N</i> =1 <i>V</i> | Space vertically. The request provides spaces in either direction. If <i>N</i> is negative, the motion is backward (upward) and is limited to the distance to the top of the page. Forward (downward) motion is truncated to the distance to the nearest trap. The space indicator ( <i>N</i> ) is ignored if not specified in the request. If the no-space mode is on, no spacing occurs (see .ns and .rs). The request causes a break.                                                 |
| .sv <i>N</i> | —               | <i>N</i> =1 <i>V</i> | Save a contiguous vertical block of size <i>N</i> . If the distance to the next trap is greater than <i>N</i> , <i>N</i> vertical spaces are output. If the distance to the next trap is less than <i>N</i> , no vertical space is immediately output; but <i>N</i> is remembered for later output (see .os). Subsequent .sv requests overwrite any still remembered <i>N</i> . No-space mode has no effect. The vertical block size indicator ( <i>N</i> ) is ignored if not specified. |
| .vs <i>N</i> | 1/6in;<br>12pts | previous             | Set vertical base-line spacing size <i>V</i> . Transient extra vertical spaces are available with \x' <i>N</i> '. The vertical base-line spacing indicator ( <i>N</i> ) is ignored if not specified in the request. Relevant parameters are a part of the current environment.                                                                                                                                                                                                           |
| Blank line   | —               | —                    | This condition causes a break and output of a blank line (just as does .sp 1).                                                                                                                                                                                                                                                                                                                                                                                                           |

\*Values separated by “;” are for the nroff and troff formatters, respectively.



TABLE 3.1

## LINE LENGTH AND INDENTING REQUESTS

| REQUEST FORM | INITIAL VALUE | IF NO ARGUMENT | EXPLANATION                                                                                                                                                                                                                                                                                                                                                            |
|--------------|---------------|----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| .in $\pm N$  | $N=0$         | previous       | Indent. The indent is set to $\pm N$ and prepended to each output line. Indent indicator ( $N$ ) is ignored if not specified by the request. Relevant parameters are a part of the current environment. The request causes a break.                                                                                                                                    |
| .ll $\pm N$  | 6.5 in        | previous       | Line length. The line length is set to $\pm N$ . In the <i>troff</i> formatter, the maximum (line-length) + (page-offset) is about 7.54 inches. The line length indicator ( $N$ ) is ignored if not specified in the request. Relevant parameters are a part of the current environment.                                                                               |
| .ti $\pm N$  | —             | ignored        | Temporary indent. The next output text line will be indented a distance $\pm N$ with respect to the current indent. The resulting total indent may not be negative. The current indent is not changed. The indent indicator ( $N$ ) is ignored if not specified in the request. Relevant parameters are a part of the current environment. The request causes a break. |



TABLE 3.J

## MACROS, STRINGS, DIVERSIONS, AND POSITION TRAPS REQUESTS

| REQUEST FORM         | INITIAL VALUE | IF NO ARGUMENT | EXPLANATION                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|----------------------|---------------|----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| .am <i>xx yy</i>     | —             | .yy=..         | Append to macro <i>xx</i> (append version of .de).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| .as <i>xx string</i> | —             | ignored        | Append <i>string</i> to string <i>xx</i> (append version of .ds).                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| .ch <i>xx N</i>      | —             | —              | Change trap location. Change the trap position for macro <i>xx</i> to be <i>N</i> . In the absence of <i>N</i> , the trap, if any, is removed. The trap location indicator ( <i>N</i> ) is ignored if not specified in the request.                                                                                                                                                                                                                                                                                                         |
| .da <i>xx</i>        | —             | end            | Divert and append to macro <i>xx</i> (append version of the .di request). Mode or relevant parameters are associated with current diversion level.                                                                                                                                                                                                                                                                                                                                                                                          |
| .de <i>xx yy</i>     | —             | .yy=..         | Define or redefine macro <i>xx</i> . The contents of the macro begin on the next input line. Input lines are copied in copy mode until the definition is terminated by a line beginning with .yy. The macro <i>yy</i> is then called. In the absence of <i>yy</i> , the definition is terminated by a line beginning with "..". A macro may contain .de requests provided the terminating macros differ or the contained definition terminator is concealed; ".." can be concealed as "\\." which will copy as "\\." and be reread as "..". |
| .di <i>xx</i>        | —             | end            | Divert output to macro <i>xx</i> . Normal text processing occurs during diversion except that page offsetting is not done. The diversion ends when the request .di or .da is encountered without an argument; extraneous requests of this type should not appear when nested diversions are being used. Mode or relevant parameters are associated with current diversion level.                                                                                                                                                            |
| .ds <i>xx string</i> | —             | ignored        | Define a string <i>xx</i> containing <i>string</i> . Any initial double-quote in <i>string</i> is stripped to permit initial blanks.                                                                                                                                                                                                                                                                                                                                                                                                        |
| .dt <i>N xx</i>      | —             | off            | Install a diversion trap at position <i>N</i> in the current diversion to invoke macro <i>xx</i> . Another .dt will redefine the diversion trap. If no arguments are given, the diversion trap is removed. Mode or relevant parameters are associated with current diversion level.                                                                                                                                                                                                                                                         |
| .em <i>xx</i>        | none          | none           | End macro. Macro <i>xx</i> will be invoked when all input has ended. The effect is the same as if the contents of <i>xx</i> had been at the end of the last file processed.                                                                                                                                                                                                                                                                                                                                                                 |



TABLE 3.J (Contd)

## MACROS, STRINGS, DIVERSIONS, AND POSITION TRAPS REQUESTS

| REQUEST FORM            | INITIAL VALUE | IF NO ARGUMENT | EXPLANATION                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|-------------------------|---------------|----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| .it <i>N</i> <i>xx</i>  | —             | off            | Input-line-count trap. An input-line-count trap is set to invoke the macro <i>xx</i> after <i>N</i> lines of text input have been read (control or request lines do not count). Text may be in-line or interpolated by in-line or trap-invoked macros. Relevant parameters are a part of the current environment.                                                                                                                           |
| .rm <i>xx</i>           | —             | ignored        | Remove. A request, macro, or string is removed. The name <i>xx</i> is removed from the name list and any related storage space is freed. Subsequent references have no effect.                                                                                                                                                                                                                                                              |
| .rn <i>xx</i> <i>yy</i> | —             | ignored        | Rename. Rename request, macro, or string from <i>xx</i> to <i>yy</i> . If <i>yy</i> exists, it is first removed.                                                                                                                                                                                                                                                                                                                            |
| .wh <i>N</i> <i>xx</i>  | —             | —              | When. A location trap is set to invoke <i>xx</i> at page position <i>N</i> ; a negative <i>N</i> is interpreted with respect to the page bottom. Any macro previously planted at <i>N</i> is replaced by <i>xx</i> . A zero <i>N</i> refers to the top of a page. In the absence of <i>xx</i> , the first found trap at <i>N</i> , if any, is removed. The page position indicator ( <i>N</i> ) is ignored if not specified in the request. |



TABLE 3.K

## PREDEFINED GENERAL NUMBER REGISTERS

| REGISTER NAME | DESCRIPTION                                                                                                                                   |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| %             | Current page number.                                                                                                                          |
| ct            | Character type (set by width function).                                                                                                       |
| dl            | Width (maximum) of last completed diversion.                                                                                                  |
| dn            | Height (vertical size) of last completed diversion.                                                                                           |
| dw            | Current day of the week (1 through 7).                                                                                                        |
| dy            | Current day of the month (1 through 31).                                                                                                      |
| hp            | Current horizontal place on input line.                                                                                                       |
| ln            | Output line number.                                                                                                                           |
| mo            | Current month (1 through 12).                                                                                                                 |
| nl            | Vertical position of last printed text base line.                                                                                             |
| sb            | Depth of string below base line (generated by width function).                                                                                |
| st            | Height of string above base line (generated by width function).                                                                               |
| yr            | Last two digits of current year.                                                                                                              |
| c.            | Provides general register access to the input line number in the current input file.<br>Contains the same value as the read-only .c register. |
| .R            | Number of number registers that remain available for use.                                                                                     |
| .b            | Emboldening factor of the current font.                                                                                                       |



TABLE 3.L  
PREDEFINED READ-ONLY NUMBER REGISTERS

| REGISTER NAME | DESCRIPTION                                                                                                                                          |
|---------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| .\$           | Number of arguments available at the current macro level.                                                                                            |
| .A            | Set to 1 in the <b>troff</b> formatter if <b>-a</b> option used; always 1 in the <b>nroff</b> formatter.                                             |
| .F            | Value is a <i>string</i> that is the name of the current input file.                                                                                 |
| .H            | Available horizontal resolution in basic units.                                                                                                      |
| .L            | Contains the current line spacing parameter (the value of the most recent <b>.ls</b> request).                                                       |
| .P            | Contains the value 1 if the current page is being printed and the value 0 if the current page is not in the <b>-o</b> option list.                   |
| .T            | Set to 1 in the <b>nroff</b> formatter if <b>-T</b> option used; always 0 in the <b>troff</b> formatter.                                             |
| .V            | Available vertical resolution in basic units.                                                                                                        |
| .a            | Post-line extra line space most recently utilized using <b>x'N'</b> .                                                                                |
| .c            | Number of lines read from current input file.                                                                                                        |
| .d            | Current vertical place in current diversion; equal to nl if no diversion.                                                                            |
| .f            | Current font as physical quadrant (1 through 4).                                                                                                     |
| .h            | Text base-line high-water mark on current page or diversion.                                                                                         |
| .i            | Current ident.                                                                                                                                       |
| .j            | Indicates the current adjustment mode and type. Can be saved and later given to the <b>.ad</b> request to restore a previous mode.                   |
| .k            | Contains the horizontal size of the text portion (without ident) of the current partially collected output line, if any, in the current environment. |
| .l            | Current line length.                                                                                                                                 |
| .n            | Length of text portion on previous output line.                                                                                                      |
| .o            | Current page offset.                                                                                                                                 |
| .p            | Current page length.                                                                                                                                 |
| .s            | Current point size.                                                                                                                                  |
| .t            | Distance to the next trap.                                                                                                                           |
| .u            | Equal to 1 in fill mode and 0 in no-fill mode.                                                                                                       |
| .v            | Current vertical line spacing.                                                                                                                       |
| .w            | Width of previous character.                                                                                                                         |
| .x            | Reserved version-dependent register.                                                                                                                 |
| .y            | Reserved version-dependent register.                                                                                                                 |
| .z            | Name of current diversion.                                                                                                                           |



TABLE 3.M

## NUMBER REGISTERS REQUESTS

| REQUEST FORM     | INITIAL VALUE                    | IF NO ARGUMENT | EXPLANATION                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |          |                    |   |                 |     |                             |   |                     |   |                     |   |                                  |   |                                  |
|------------------|----------------------------------|----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------|--------------------|---|-----------------|-----|-----------------------------|---|---------------------|---|---------------------|---|----------------------------------|---|----------------------------------|
| .af <i>R c</i>   | arabic                           | —              | <p>Assign format. Format <i>c</i> is assigned to register <i>R</i>. Available formats are:</p> <table><tr><th><i>c</i></th><th>NUMBERING SEQUENCE</th></tr><tr><td>1</td><td>0,1,2,3,4,5,...</td></tr><tr><td>001</td><td>000,001,002,003,004,005,...</td></tr><tr><td>i</td><td>0,i,ii,iii,iv,v,...</td></tr><tr><td>I</td><td>0,I,II,III,IV,V,...</td></tr><tr><td>a</td><td>0,a,b,...,z,aa,ab,...,zz,aaa,...</td></tr><tr><td>A</td><td>0,A,B,...,Z,AA,AB,...,ZZ,AAA,...</td></tr></table> <p>An arabic format having <i>N</i> digits specifies a field width of <i>N</i> digits. Read-only registers and width function are always arabic.</p> | <i>c</i> | NUMBERING SEQUENCE | 1 | 0,1,2,3,4,5,... | 001 | 000,001,002,003,004,005,... | i | 0,i,ii,iii,iv,v,... | I | 0,I,II,III,IV,V,... | a | 0,a,b,...,z,aa,ab,...,zz,aaa,... | A | 0,A,B,...,Z,AA,AB,...,ZZ,AAA,... |
| <i>c</i>         | NUMBERING SEQUENCE               |                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |          |                    |   |                 |     |                             |   |                     |   |                     |   |                                  |   |                                  |
| 1                | 0,1,2,3,4,5,...                  |                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |          |                    |   |                 |     |                             |   |                     |   |                     |   |                                  |   |                                  |
| 001              | 000,001,002,003,004,005,...      |                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |          |                    |   |                 |     |                             |   |                     |   |                     |   |                                  |   |                                  |
| i                | 0,i,ii,iii,iv,v,...              |                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |          |                    |   |                 |     |                             |   |                     |   |                     |   |                                  |   |                                  |
| I                | 0,I,II,III,IV,V,...              |                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |          |                    |   |                 |     |                             |   |                     |   |                     |   |                                  |   |                                  |
| a                | 0,a,b,...,z,aa,ab,...,zz,aaa,... |                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |          |                    |   |                 |     |                             |   |                     |   |                     |   |                                  |   |                                  |
| A                | 0,A,B,...,Z,AA,AB,...,ZZ,AAA,... |                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |          |                    |   |                 |     |                             |   |                     |   |                     |   |                                  |   |                                  |
| .nr <i>R +NM</i> |                                  | —              | <p>Number register. The number register <i>R</i> is assigned the value <i>+N</i> with respect to the previous value, if any. The automatic incrementing value is set to <i>M</i>. The number register value (<i>N</i>) is ignored if not specified in the request.</p>                                                                                                                                                                                                                                                                                                                                                                             |          |                    |   |                 |     |                             |   |                     |   |                     |   |                                  |   |                                  |
| .rr <i>R</i>     | —                                | —              | <p>Remove register. The number register <i>R</i> is removed. If many registers are being created dynamically, it may be necessary to remove registers that are no longer used in order to recapture internal storage space for newer registers.</p>                                                                                                                                                                                                                                                                                                                                                                                                |          |                    |   |                 |     |                             |   |                     |   |                     |   |                                  |   |                                  |



TABLE 3.N

## TABS, LEADERS, AND FIELDS REQUESTS

| REQUEST FORM     | INITIAL VALUE* | IF NO ARGUMENT | EXPLANATION                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|------------------|----------------|----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| .fc <i>a b</i>   | off            | off            | Field delimiter is set to <i>a</i> . The padding indicator is set to the space character or to <i>b</i> , if given. In the absence of arguments, the field mechanism is turned off.                                                                                                                                                                                                                                                                                                                             |
| .lc <i>c</i>     | .              | none           | Leader repetition character becomes <i>c</i> or is removed specifying motion. Relevant parameters are a part of the current environment.                                                                                                                                                                                                                                                                                                                                                                        |
| .ta <i>Nt...</i> | 8n; 0.5 in     | none           | Set tab stops and types.<br>When the tab type indicator ( <i>t</i> ) is:<br>R — text is right adjusted<br>C — text is centered<br>Absent — text is left adjusted.<br>Tab stops for the <b>troff</b> formatter are preset every 0.5 inch; tab stops for the <b>nroff</b> formatter are preset every eight nominal character widths. Stop values are separated by spaces, and a value preceded by + is treated as an increment to the previous stop value. Relevant parameters are a part of current environment. |
| .tc <i>c</i>     | none           | none           | Tab repetition character becomes <i>c</i> or is removed specifying motion. Relevant parameters are a part of the current environment.                                                                                                                                                                                                                                                                                                                                                                           |

\*Values separated by “;” are for the **nroff** and **troff** formatters, respectively.



TABLE 3.0

## INPUT AND OUTPUT CONVENTIONS AND CHARACTER TRANSLATIONS REQUESTS

| REQUEST FORM       | INITIAL VALUE* | IF NO ARGUMENT | EXPLANATION                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|--------------------|----------------|----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| .cc <i>c</i>       | .              | .              | Set control character to <i>c</i> or reset to ".". Relevant parameters are a part of the current environment.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| .cu <i>N</i>       | off            | <i>N</i> =1    | Continuous underline in the <i>nroff</i> formatter. A variant of .ul that causes every character to be underlined. Identical to .ul in the <i>troff</i> formatter. Relevant parameters are a part of the current environment.                                                                                                                                                                                                                                                                                                                                                                                       |
| .c2 <i>c</i>       | '              | '              | Set no-break control character to <i>c</i> or reset to "'". Relevant parameters are a part of the current environment.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| .ec <i>c</i>       | \              | \              | Set escape character to \ or to <i>c</i> , if given.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| .eo                | on             | —              | Turn escape character mechanism off.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| .lg <i>N</i>       | of;on          | on             | Ligature mode is turned on if <i>N</i> is absent or nonzero and turned off if <i>N</i> =0. If <i>N</i> =2, only the 2-character ligatures are automatically invoked. Ligature mode is inhibited for request, macro, string, register, file names, and copy mode. There is no effect in the <i>nroff</i> formatter.                                                                                                                                                                                                                                                                                                  |
| .tr <i>abcd...</i> | none           | —              | Translate <i>a</i> into <i>b</i> , <i>c</i> into <i>d</i> , etc. on output. If an odd number of characters is given, the last one will be mapped into the space character. To be consistent, a particular translation must stay in effect from input to output time. Initially, there are no translate values.                                                                                                                                                                                                                                                                                                      |
| .uf <i>F</i>       | Italic         | Italic         | Underline font set to <i>F</i> (to be switched to by .ul). In the <i>nroff</i> formatter, <i>F</i> may not be on position 1 (initially Times Roman).                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| .ul <i>N</i>       | off            | <i>N</i> =1    | Underline in the <i>nroff</i> formatter (italicize in <i>troff</i> the next <i>N</i> input text lines). Switch to underline font saving the current font for later restoration; other font changes within the span of a .ul will take effect, but the restoration will undo the last change. Output generated by a .ul is affected by the font change but does not decrement <i>N</i> . If <i>N</i> is greater than 1, there is the risk that a trap interpolated macro may provide text lines within the span, which environment switching can prevent. Relevant parameters are a part of the current environment. |

\*Values separated by ";" are for the *nroff* and *troff* formatters, respectively.



TABLE 3.P

## LOCAL MOTIONS

## VERTICAL LOCAL MOTION

| FUNCTION           | EFFECT IN              |               |
|--------------------|------------------------|---------------|
|                    | TROFF                  | NROFF         |
| <code>\ v'N</code> | Move distance <i>N</i> |               |
| <code>\ u</code>   | 1/2 em up              | 1/2 line up   |
| <code>\ d</code>   | 1/2 em down            | 1/2 line down |
| <code>\ r</code>   | 1 em up                | 1 line up     |

## HORIZONTAL LOCAL MOTION

| FUNCTION               | EFFECT IN                   |         |
|------------------------|-----------------------------|---------|
|                        | TROFF                       | NROFF   |
| <code>\ h'N</code>     | Move distance <i>N</i>      |         |
| <code>\ (space)</code> | Unpaddable space-size space |         |
| <code>\ 0</code>       | Digit-size space            |         |
| <code>\ !</code>       | 1/6 em space                | ignored |
| <code>\ ^</code>       | 1/12 em space               | ignored |



TABLE 3.Q

## HYPHENATION REQUESTS

| REQUEST FORM         | INITIAL VALUE    | IF NO ARGUMENT  | EXPLANATION                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|----------------------|------------------|-----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| .hc <i>c</i>         | \%               | \%              | Hyphenation character. Hyphenation indicator character is set to <i>c</i> or to the default \%. The indicator does not appear in the output. Relevant parameters are a part of the current environment.                                                                                                                                                                                                                                          |
| .hw <i>word l...</i> | —                | ignored         | Exception words. Hyphenation points in words are specified with imbedded minus signs. Versions of a word that end with an <i>s</i> are implied; i.e., <i>dig-it</i> implies <i>dig-its</i> . This list is examined initially and after each suffix stripping. Space available is small—about 128 characters.                                                                                                                                     |
| .hy <i>N</i>         | off, <i>N</i> =0 | on, <i>N</i> =1 | Hyphenate. Automatic hyphenation is turned on for <i>N</i> greater than or equal to 1 or off for <i>N</i> =0. If <i>N</i> =2, last lines (ones that will cause a trap) are not hyphenated. For <i>N</i> =4 and <i>N</i> =8, the last and first two characters, respectively, of a word are not divided. These values are additive; i.e., <i>N</i> =14 invokes all three restrictions. Relevant parameters are a part of the current environment. |
| .nh                  | no hyphen        | —               | No hyphenation. Automatic hyphenation is turned off. Relevant parameters are a part of the current environment.                                                                                                                                                                                                                                                                                                                                  |



TABLE 3.R

## THREE-PART TITLES REQUEST

| REQUEST FORM                                       | INITIAL VALUE | IF NO ARGUMENT | EXPLANATION                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|----------------------------------------------------|---------------|----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| .lt $\pm N$                                        | 6.5 in        | previous       | Length of title set to $\pm N$ . Line length and title length are independent. Indents do not apply to titles; page offsets do. Relevant parameters are a part of the current environment.                                                                                                                                                                                                                                                                   |
| .pc <i>c</i>                                       | %             | off            | Page number character set to <i>c</i> or removed. The page number register remains %.                                                                                                                                                                                                                                                                                                                                                                        |
| .tl ' <i>left</i> ' <i>center</i> ' <i>right</i> ' |               |                | Three-part title. The strings <i>left</i> , <i>center</i> , and <i>right</i> are, respectively, left adjusted, centered, and right adjusted in the current title length. Any of the strings may be empty, and overlapping is permitted. If the page number character (initially %) is found within any of the fields, it is replaced by the current page number having the format assigned to register %. Any character may be used as the string delimiter. |



TABLE 3.5

## OUTPUT LINE NUMBERING REQUESTS

| REQUEST FORM      | INITIAL VALUE | IF NO ARGUMENT | EXPLANATION                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|-------------------|---------------|----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| .nm $\pm N M S I$ | —             | off            | Line number mode. If $\pm N$ is given, line numbering is turned on; and the next output line is numbered $\pm N$ . Default values are $M=1$ , $S=1$ , and $I=0$ . Parameters corresponding to missing arguments are unaffected; a nonnumeric argument is considered missing. In the absence of all arguments, numbering is turned off, and the next line number is preserved for possible further use in number register $ln$ . Relevant parameters are a part of the current environment. |
| .nn $N$           | —             | $N=1$          | Next $N$ lines are not numbered. Relevant parameters are a part of the current environment.                                                                                                                                                                                                                                                                                                                                                                                                |



TABLE 3.T

## CONDITIONAL ACCEPTANCE OF INPUT REQUESTS

| REQUEST FORM                             | INITIAL VALUE | IF NO ARGUMENT | EXPLANATION                                                                                                 |
|------------------------------------------|---------------|----------------|-------------------------------------------------------------------------------------------------------------|
| <i>.el anything</i>                      |               | —              | The "else" portion of "if-else".                                                                            |
| <i>.ie c anything</i>                    |               | —              | The "if" portion of "if-else". The <i>c</i> can be any of the forms acceptable with the <i>.if</i> request. |
| <i>.if c anything</i>                    |               | —              | If condition <i>c</i> true, accept <i>anything</i> as input; for multi-line case, use <i>\{anything\}</i> . |
| <i>.if !c anything</i>                   |               | —              | If condition <i>c</i> false, accept <i>anything</i> .                                                       |
| <i>.if N anything</i>                    |               | —              | If expression <i>N</i> greater than 0, accept <i>anything</i> .                                             |
| <i>.if !N anything</i>                   |               | —              | If expression <i>N</i> less than or equal to 0, accept <i>anything</i> .                                    |
| <i>.if 'string1' string2' anything</i>   |               |                | If <i>string 1</i> is identical to <i>string 2</i> , accept <i>anything</i> .                               |
| <i>.if !'string 1' string2' anything</i> |               |                | If <i>string 1</i> is not identical to <i>string 2</i> , accept <i>anything</i> .                           |

TABLE 3.U

## ENVIRONMENT SWITCHING REQUEST

| REQUEST FORM | INITIAL VALUE | IF NO ARGUMENT | EXPLANATION                                                                                                                                                                     |
|--------------|---------------|----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>.ev N</i> | <i>N=0</i>    | previous       | Environment switched to 0, 1, or 2. Switching is done in pushdown fashion so that restoring a previous environment must be done with <i>.ev</i> rather than specific reference. |



TABLE 3.V

## INSERTIONS FROM STANDARD INPUT REQUESTS

| REQUEST FORM                   | INITIAL VALUE | IF NO ARGUMENT          | EXPLANATION                                                                                                                                                                                                                                                                   |
|--------------------------------|---------------|-------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>.ex</code>               | —             | —                       | Exit from the <code>nroff/troff</code> formatter. Text processing is terminated exactly as if all input had ended.                                                                                                                                                            |
| <code>.rd <i>prompt</i></code> | —             | <code>prompt=BEL</code> | Read insertion from the standard input until two newline characters in a row are found. If standard input is the user keyboard, a <i>prompt</i> (or a BEL) is written onto the user terminal. The request behaves like a macro; arguments may be placed after <i>prompt</i> . |

TABLE 3.W

## INPUT/OUTPUT FILE SWITCHING REQUESTS

| REQUEST FORM                     | IF NO ARGUMENT | EXPLANATION                                                                                                                                                                                                                                                                             |
|----------------------------------|----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>.nx <i>filename</i></code> | end-of-file    | Next file is <i>filename</i> . The current file is considered ended, and the input is immediately switched to <i>filename</i> .                                                                                                                                                         |
| <code>.pi <i>program</i></code>  | —              | Pipe output to <i>program</i> ( <code>nroff</code> formatter only). This request must occur before any printing occurs. No arguments are transmitted to <i>program</i> .                                                                                                                |
| <code>.so <i>filename</i></code> | —              | Switch source file (pushdown). The top input level (file reading) is switched to <i>filename</i> . Contents are interpolated at the point the request is encountered. When the new file ends, input is again taken from the original file. The <code>.so</code> requests may be nested. |



TABLE 3.X

## MISCELLANEOUS REQUESTS

| REQUEST FORM | INITIAL VALUE | IF NO ARGUMENT | EXPLANATION                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|--------------|---------------|----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| .co          | —             | —              | Specify the point in the macro file at which compaction ends. When <i>-kname</i> is called on the command line, all lines in the file <i>name</i> before the .co request will be compacted.                                                                                                                                                                                                                                                                                                                                                               |
| .fl          | —             | —              | Flush output buffer. Used in interactive debugging to force output. The request causes a break.                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| .ig yy       | —             | .yy=..         | Ignore input lines until call of <i>yy</i> . This request behaves like the .de request except that the input is discarded. The input is read in <i>copy</i> mode, and any automatically incremented registers will be affected.                                                                                                                                                                                                                                                                                                                           |
| .mc c N      | —             | off            | Sets margin character <i>c</i> and separation <i>N</i> . Specifies that a margin character <i>c</i> appear a distance <i>N</i> to the right of the right margin after each nonempty text line (except those produced by .fl). If the output line is too long (as can happen in no-fill mode), the character will be appended to the line. If <i>N</i> is not given, the previous <i>N</i> is used; the initial <i>N</i> is 0.2 inches in the <i>nroff</i> formatter and 1 em in <i>troff</i> . Relevant parameters are a part of the current environment. |
| .pm t        | —             | all            | Print macros. The names and sizes of all defined macros and strings are printed on the user terminal. If <i>t</i> is given, only the total of the sizes is printed. Sizes are given in blocks of 128 characters.                                                                                                                                                                                                                                                                                                                                          |
| .tm string   | —             | newline        | Print <i>string</i> on terminal (UNIX operating system standard message output). After skipping initial blanks, string (rest of the line) is read in copy mode and written on the user terminal.                                                                                                                                                                                                                                                                                                                                                          |

TABLE 3.Y

## OUTPUT AND ERROR MESSAGES REQUEST

| REQUEST FORM | INITIAL VALUE | IF NO ARGUMENT | EXPLANATION                                                                                                                                                                                                 |
|--------------|---------------|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| .ab text     | —             | —              | Prints <i>text</i> on the message output and terminates without further processing. If <i>text</i> is missing, "User Abort." is printed. This request does not cause a break. The output buffer is flushed. |



TABLE 3.Z

NAMING CONVENTION FOR THE  
MATHEMATICS TYPESETTING PROGRAM

| CHARACTER<br>SEQUENCE | OUTPUT        | INPUT<br>NAME | CHARACTER  |
|-----------------------|---------------|---------------|------------|
| >=                    | $\geq$        | DELTA         | $\Delta$   |
| <=                    | $\leq$        | GAMMA         | $\Gamma$   |
| =                     | $\equiv$      | LAMBDA        | $\Lambda$  |
| !=                    | $\neq$        | OMEGA         | $\Omega$   |
| + -                   | $\pm$         | PHI           | $\Phi$     |
| - >                   | $\rightarrow$ | PI            | $\Pi$      |
| < -                   | $\leftarrow$  | PSI           | $\Psi$     |
| <<                    | $\ll$         | SIGMA         | $\Sigma$   |
| >>                    | $\gg$         | THETA         | $\Theta$   |
| inf                   | $\infty$      | UPSILON       | $\Upsilon$ |
| partial               | $\partial$    | XI            | $\Xi$      |
| half                  | $\frac{1}{2}$ | alpha         | $\alpha$   |
| prime                 | $'$           | beta          | $\beta$    |
| approx                | $\approx$     | chi           | $\chi$     |
| nothing               |               | delta         | $\delta$   |
| cdot                  | $\cdot$       | epsilon       | $\epsilon$ |
| times                 | $\times$      | eta           | $\eta$     |
| del                   | $\nabla$      | gamma         | $\gamma$   |
| grad                  | $\nabla$      | iota          | $\iota$    |
| ...                   | $\dots$       | kappa         | $\kappa$   |
| ,...,                 | $\dots,$      | lambda        | $\lambda$  |
| sum                   | $\Sigma$      | mu            | $\mu$      |
| int                   | $\int$        | nu            | $\nu$      |
| prod                  | $\prod$       | omega         | $\omega$   |
| union                 | $\cup$        | omicron       | $\circ$    |
| inter                 | $\cap$        | phi           | $\phi$     |
|                       |               | pi            | $\pi$      |
|                       |               | psi           | $\psi$     |
|                       |               | rho           | $\rho$     |
|                       |               | sigma         | $\sigma$   |
|                       |               | tau           | $\tau$     |
|                       |               | theta         | $\theta$   |
|                       |               | upsilon       | $\upsilon$ |
|                       |               | xi            | $\xi$      |
|                       |               | zeta          | $\zeta$    |







## IV. MEMORANDUM MACROS

### 1. Introduction

#### 1.1 Purpose

This section is a guide and reference manual for users of Memorandum Macros (MM). These macros provide a general purpose package of text formatting macros for use with the UNIX operating system text formatters **nroff** and **troff** (refer to **troff(1)** in the User's Manual—UNIX Operating System for more details). A reference of the form **name(N)** points to page **name** in section **N** of the User's Manual.

#### 1.2 Conventions

Each part of this section explains a single facility of MM. In general, the earlier a part occurs, the more necessary the information is for most users. Some of the later parts can be completely ignored if MM defaults are acceptable. Likewise, each part progresses from general case to special-case facilities. It is recommended that a user read a part in detail only to the point where there is enough information to obtain the desired format, then skim the rest of the part because some details may be of use to only a few.

Numbers enclosed in curly brackets ({} ) refer to paragraph numbers within this section. For example, this is paragraph {1.2}.

In the synopses of macro calls, square brackets ( [ ] ) surrounding an argument indicate that it is optional. Ellipses (...) show that the preceding argument may appear more than once.

Figure 4.1 shows both **nroff** and **troff** formatter outputs (of files using MM macros) for a simple letter. In those cases in which the behavior of the two formatters is obviously different, the **nroff** formatter output is described first with the **troff** formatter output following in parentheses. For example:

The title is underlined (*italic*).

means that the title is underlined by the **nroff** formatter and italicized by the **troff** formatter.

#### 1.3 Document Structure

Input for a document to be formatted with the MM text formatting macro package has four major segments, any of which may be omitted; if present, the segments must occur in the following order:

- *Parameter setting segment* sets the general style and appearance of a document. The user can control page width, margin justification, numbering styles for heading and lists, page headers and footers {9}, and many other properties of the document. Also, the user can add macros or redefine existing ones. This segment can be omitted entirely if the user is satisfied with default values; it produces no actual output, but performs only the formatter setup for the rest of the document.
- *Beginning segment* includes those items that occur only once, at the beginning of a document, e.g., title, author's name, date.
- *Body segment* is the actual text of the document. It may be as small as a single paragraph or as large as hundreds of pages. It may have a hierarchy of headings up to seven levels deep {4}. Headings are automatically numbered (if desired) and can be saved to generate the table of contents. Five additional levels of subordination are provided by a set of list macros for automatic numbering, alphabetic sequencing, and "marking" of list items {5}. The body may also contain various types of displays, tables, figures, references, and footnotes {7, 8, 11}.
- *Ending segment* contains those items that occur only once at the end of a document. Included are signature(s) and lists of notations (e.g., "Copy to" lists) {6.11}. Certain macros may be invoked here to print



information that is wholly or partially derived from the rest of the document, such as the table of contents or the cover sheet for a document {10}.

Existence and size of these four segments varies widely among different document types. Although a specific item (such as date, title, author names, etc.) may differ depending on the document, there is a uniform way of typing it into an input text file.

#### 1.4 Input Text Structure

In order to make it easy to edit or revise input file text at a later time.

- Input lines should be kept short
- Lines should be broken at the end of clauses
- Each new sentence should begin on a new line.

#### 1.5 Definitions

**Formatter** refers to either the **nroff** or **troff** text-formatting program.

**Requests** are built-in commands recognized by the formatters. Although a user seldom needs to use these requests directly {3.10}, this section contains references to some of the requests. For example, the request

`.sp`

inserts a blank line in the output at the place the request occurs in the input text file.

**Macros** are named collections of requests. Each macro is an abbreviation for a collection of requests that would otherwise require repetition. The MM package supplies many macros, and the user can define additional ones. Macros and requests share the same set of names and are used in the same way.

Table 4.A is an alphabetical list of macro names used by MM. The first line of each item lists the name of the macro, a brief description, and a reference to the paragraph in which the macro is described. The second line illustrates a typical call of the macro.

**Strings** provide character variables, each of which names a string of characters. Strings are often used in page headers, page footers, and lists. These registers share the pool of names used by requests and macros. A string can be given a value via the `.ds` (define string) request; and its value can be obtained by referencing its name, preceded by `"\"` (for 1-character names) or `"\"` (for 2-character names). For instance, the string `DT` in MM normally contains the current date, thus the input line

Today is `\*(DT`.

may result in the following output:

Today is December 16,1981.

The current date can be replaced, e.g.:

`.ds DT 01/01/79`

by invoking a macro designed for that purpose {6.8}. Table 4.B is an alphabetical list of string names used by MM. A brief description, paragraph reference, and initial (default) value(s) are given for each.



Number registers fill the role of integer variables. These registers are used for flags and for arithmetic and automatic numbering. A register can be given a value using a `.nr` request and be referenced by preceding its name by `\n` (for 1-character names) or `\n(` (for 2-character names). For example, the following sets the value of the register `d` to one more than that of the register `dd`:

```
.nr d 1+\n(dd
```

Table 4.C is an alphabetical list of number register names giving for each a brief description, paragraph reference, initial (default) value, and legal range of values (where `[m:n]` means values from `m` to `n`, inclusive).

Paragraph 14.1 contains naming conventions for requests, macros, strings, and number registers. Table 4.A, 4.B, and 4.C list all macros, strings, and number registers defined in MM.

## 2. Usage

This part describes how to access MM, illustrates UNIX operating system command lines appropriate for various output devices, and describes command line flags for the MM text formatting macro package.

### 2.1 The mm Command

The `mm(1)` command can be used to prepare documents using the `nroff` formatter and MM; this command invokes `nroff` with the `-cm` flag {2.2}. The `mm` command has options to specify preprocessing by `tbl` and/or by `neqn` and for postprocessing by various output filters. Any arguments or flags that are not recognized by the `mm` command, e.g., `-rC3`, are passed to the `nroff` formatter or to MM, as appropriate. Options, which can occur in any order but must appear before the file names, are:

| OPTION                 | MEANING                                                                                                                                                                 |
|------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-e</code>        | The <code>neqn</code> is to be invoked; also causes <code>neqn</code> to read <code>/usr/pub/eqnchar</code> [see <code>eqnchar(7)</code> ].                             |
| <code>-t</code>        | The <code>tbl(1)</code> is to be invoked.                                                                                                                               |
| <code>-c</code>        | The <code>col(1)</code> is to be invoked.                                                                                                                               |
| <code>-E</code>        | The <code>-e</code> option of the <code>nroff</code> formatter is to be invoked.                                                                                        |
| <code>-y</code>        | The <code>-mm</code> (uncompacted macros) is to be used instead of <code>-cm</code> .                                                                                   |
| <code>-12</code>       | The 12-pitch mode is to be used. The pitch switch on the terminal should be set to 12 if necessary.                                                                     |
| <code>-T450</code>     | Output is to a DASI 450. This is the default terminal type [unless <code>\$TERM</code> is set; see <code>sh(1)</code> ]. It is also equivalent to <code>-T1620</code> . |
| <code>-T450-12</code>  | Output is to a DASI 450 in 12-pitch mode.                                                                                                                               |
| <code>-T300</code>     | Output is to a DASI 300 terminal.                                                                                                                                       |
| <code>-T300-12</code>  | Output is to a DASI 300 in 12-pitch mode.                                                                                                                               |
| <code>-T300s</code>    | Output is to a DASI 300S.                                                                                                                                               |
| <code>-T300s-12</code> | Output is to a DASI 300S in 12-pitch mode.                                                                                                                              |
| <code>-T4014</code>    | Output is to a Tektronix 4014.                                                                                                                                          |



| OPTION  | MEANING                                                                                                                                        |
|---------|------------------------------------------------------------------------------------------------------------------------------------------------|
| -T37    | Output is to a TELETYPE® Model 37.                                                                                                             |
| -T382   | Output is to a DTC-382.                                                                                                                        |
| -T4000a | Output is to a Trendata 4000A.                                                                                                                 |
| -TX     | Output is prepared for an EBCDIC line printer.                                                                                                 |
| -Thp    | Output is to an HP264x (implies -c).                                                                                                           |
| -T43    | Output is to a TELETYPE® Model 43 (implies -c).                                                                                                |
| -T40/4  | Output is to a TELETYPE® Model 40/4 (implies -c).                                                                                              |
| -T745   | Output is to a Texas Instrument 700 series terminal (implies -c).                                                                              |
| -T2631  | Output is prepared for an HP2631 printer where -T2631-e and -T2631-c may be used for expanded and compressed modes, respectively (implies -c). |
| -Tlp    | Output is to a device with no reverse or partial line motions or other special features (implies -c).                                          |

Any other -T option given does not produce an error; it is equivalent to -Tlp.

A similar command is available for use with the troff formatter [ mmt(1)].

## 2.2 The -cm or -mm Flag

The MM package can also be invoked by including the -cm or -mm flag as an argument to the formatter. The -cm flag causes the precompact version of the macros to be loaded. The -mm flag causes the file /usr/lib/tmac/tmac.m to be read and processed before any other files. This action defines the MM macros, sets default values for various parameters, and initializes the formatter to be ready to process the input text files.

## 2.3 Typical Command Lines

The prototype command lines are as follows (with the various options explained in {2.4}):

- Text without tables or equations:

```
mm [options] file ...  
or  
nroff [options] -cm file ...
```

```
mmt [options] file ...  
or  
troff [options] -cm file ...
```

- Text with tables:

```
mm -t [options] file ...  
or  
tbl file ... | nroff [options] -cm
```



```
mmt -t [options] file ...
or
tbl file ... | troff [options] -cm
```

- Text with equations:

```
mm -e [options] file ...
or
neqn /usr/pub/eqnchar file ... | nroff [options] -cm
```

```
mmt -e [options] file ...
or
eqn /usr/pub/eqnchar file ... | troff [options] -cm
```

- Text with both tables and equations:

```
mm -t -e [options] file ...
or
tbl file ... | neqn /usr/pub/eqnchar -- | nroff [options] -cm
```

```
mmt -t -e [options] file ...
or
tbl file ... | eqn /usr/pub/eqnchar -- | troff \ [options] -cm
```

When formatting a document with the **nroff** processor, the output should normally be processed for a specific type of terminal because the output may require some features that are specific to a given terminal, e.g., reverse paper motion or half-line paper motion in both directions. Some commonly used terminal types and the command lines appropriate for them are given below. More information is found in paragraph {2.4} of this part, and 300(1), 450(1), 4014(1), hp(1), col(1), termio(4), and term(5) of the User's Manual — UNIX Operating System.

- DASI 450 in 10-pitch, 6 lines/inch mode, with 0.75 inch offset, and a line length of 6 inches (60 characters) where this is the default terminal type so no **-T** option is needed (unless **\$TERM** is set to another value):

```
mm file ...
or
nroff -T450 -h -cm file ...
```

- DASI 450 in 12-pitch, 6 lines/inch mode, with 0.75 inch offset, and a line length of 6 inches (72 characters):

```
mm -12 file ...
or
nroff -T450-12 -h -cm file ...
```

or to increase the line length to 80 characters and decrease the offset to 3 characters:

```
mm -12 -rW80 -rO3 file ...
or
nroff -T450-12 -rW80 -rO3 -h -cm file ...
```

- Hewlett-Packard HP264x CRT family:

```
mm -Thp file ...
```



or  
 nroff -cm file ... | col | hp

- Any terminal incapable of reverse paper motion and also lacking hardware tab stops (Texas Instruments 700 series, etc.):

mm -T745 file ...  
 or  
 nroff -cm file ... | col -x

The `tbl(1)` and `eqn(1)/neqn` formatters, if needed, must be invoked as shown in the command lines illustrated earlier.

If 2-column processing {12.4} is used with the `nroff` formatter, either the `-c` option must be specified to `mm(1)` [`mm(1)` uses `col(1)` automatically for many terminal types {2.1}] or the `nroff` formatter output must be postprocessed by `col(1)`. In the latter case, the `-T37` terminal type must be specified to the `nroff` formatter, the `-h` option must not be specified, and the output of `col(1)` must be processed by the appropriate terminal filter [e.g., `450(1)`]; `mm(1)` with the `-c` option handles all this automatically.

## 2.4 Parameters Set From Command Line

Number registers are commonly used within MM to hold parameter values that control various aspects of output style. Many of these values can be changed within the text files with `.nr` requests. In addition, some of these registers can be set from the command line. This is a useful feature for those parameters that should not be permanently embedded within the input text. If used, the number registers (with the possible exception of the register `P` below) must be set on the command line (or before the MM macro definitions are processed). The number register meanings are:

| REGISTER          | MEANING                                                                                                                                                                                                                                                                                                                                    |
|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-rAn</code> | <code>n = 1</code> has effect of invoking the <code>.AF</code> macro without an argument {6.9} .<br><code>n = 2</code> permits use of Bell System logo, if available, on a printing device (currently available for Xerox 9700 only).                                                                                                      |
| <code>-rCn</code> | sets type of copy (e.g., DRAFT) to be printed at bottom of each page {9.2.4} .<br><code>n = 1</code> for OFFICIAL FILE COPY.<br><code>n = 2</code> for DATE FILE COPY.<br><code>n = 3</code> for DRAFT with single spacing and default paragraph style.<br><code>n = 4</code> for DRAFT with double spacing and 10-space paragraph indent. |
| <code>-rD1</code> | sets <i>debug mode</i> .<br>This flag requests formatter to continue processing even if MM detects errors that would otherwise cause termination. It also includes some debugging information in the default page header {9.2.1, 12.3} .                                                                                                   |



- | REGISTER | MEANING                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -rEn     | controls font of Subject/Date/From fields.<br>n = 0, fields are bold (default for the troff formatter).<br>n = 1, fields are Roman font (regular text-default for the nroff formatter).                                                                                                                                                                                                                                                                                                               |
| -rLk     | sets length of physical page to k lines.<br>For the nroff formatter, k is an unscaled number representing lines.<br>For the troff formatter, k must be scaled.<br>Default value is 66 lines per page.<br>This flag is used, for example, when directing output to a Versatec* printer.                                                                                                                                                                                                                |
| -rNn     | specifies page numbering style.<br>n = 0 (default), all pages get the prevailing header {9.2.1} .<br>n = 1, page header replaces footer on page 1 only.<br>n = 2, page header is omitted from page 1.<br>n = 3, "section-page" numbering {4.5} occurs (.FD {8.3} and .RP {11.4} defines footnote and reference numbering in sections).<br>n = 4, default page header is suppressed; however, a user-specified header is not affected.<br>n = 5, "section-page" and "section-figure" numbering occurs. |

| n | PAGE 1                                        | PAGES 2FF.                   |
|---|-----------------------------------------------|------------------------------|
| 0 | header                                        | header                       |
| 1 | header replaces footer                        | header                       |
| 2 | no header                                     | header                       |
| 3 | "section-page" as footer                      | same as page 1               |
| 4 | no header                                     | no header unless .PH defined |
| 5 | "section-page" as footer and "section-figure" | same as page 1               |

Contents of the prevailing header and footer do not depend on number register *N* value; *N* controls only whether the header (*N*=3) or the footer (*N*=5) is printed, as well as the page numbering style. If header and footer are null {9.2.1, 9.2.4} , the value of *N* is irrelevant.

- rOk offsets output k spaces to the right.  
For the nroff formatter, k is an unscaled number representing lines or character positions.  
For the troff formatter, k must be scaled.  
This flag is helpful for adjusting output positioning on some terminals. The default offset, if this regular is not set on the command line, is 0.75 inches.  
**Note:** Register name is the capital letter "O".

- rPn specifies that pages of the document are to be numbered starting with n.  
This register may also be set via a .nr request in the input text.
- rSn sets point size and vertical spacing for the document. The default n is 10, i.e., 10-point type on 12-point vertical spacing, giving six lines per inch {12.9} . This flag applies to the troff formatter only.

\* Registered Trademark of Versatec, Inc.



| REGISTER          | MEANING                                                                                                                                                                                                                                                                                                                                                                                                                |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-rTn</code> | <p>provides register settings for certain devices.</p> <p>If <i>n</i> is 1, line length and page offset are set to 80 and 3, respectively.</p> <p>Setting <i>n</i> to 2 changes the page length to 84 lines per page and inhibits underlining; it is meant for output sent to the Versatec printer.</p> <p>The default value for <i>n</i> is 0.</p> <p>This flag applies to the <code>nroff</code> formatter only.</p> |
| <code>-rU1</code> | <p>controls underlining of section headings.</p> <p>This flag causes only letters and digits to be underlined. Otherwise, all characters (including spaces) are underlined {4.2.2.4.2}.</p> <p>This flag applies to the <code>nroff</code> formatter only.</p>                                                                                                                                                         |
| <code>-rWk</code> | <p>sets page width (line length and title length) to <i>k</i>.</p> <p>For the <code>nroff</code> formatter, <i>k</i> is an unscaled number representing character positions.</p> <p>For the <code>troff</code> formatter, <i>k</i> must be scaled.</p> <p>This flag can be used to change page width from the default value of 6 inches (60 characters in 10 pitch or 72 characters in 12 pitch).</p>                  |

## 2.5 Omission of `-cm` or `-mm` Flag

If a large number of arguments is required on the command line, it may be convenient to set up the first (or only) input file of a document as follows:

```
zero or more initializations of registers listed in {2.4}
.so /usr/lib/tmac/tmac.m
remainder of text
```

In this case, the user must not use the `-cm` or `-mm` flag [nor the `mm(1)` or `mmt(1)` command]; the `.so` request has the equivalent effect, but registers in {2.4} must be initialized before the `.so` request because their values are meaningful only if set before macro definitions are processed. When using this method, it is best to lock into the input file only those parameters that are seldom changed. For example:

```
.nr W 80
.nr O 10
.nr N 3
.so /usr/lib/tmac/tmac.m
.H 1 "INTRODUCTION"
```

specifies, for the `nroff` formatter, a line length (W) of 80, a page offset (O) of 10, and "section-page" (N) numbering.

## 3. Formatting Concepts

### 3.1 Basic Terms

Normal action of the formatters is to fill output lines from one or more input lines. Output lines may be



justified so that both the left and right margins are aligned. As lines are being filled, words may also be hyphenated {3.4} as necessary. It is possible to turn any of these modes on and off (.SA {12.2}, Hy {3.4}, and the .nf and .fi formatter requests). Turning off fill mode also turns off justification and hyphenation.

Certain formatting commands (requests and macros) cause filling of the current output line to cease, the line (of whatever length) to be printed, and subsequent text to begin a new output line. This printing of a partially filled output line is known as a *break*. A few formatter requests and most of the MM macros cause a break.

Formatter requests {3.10} can be used with MM; however, there are consequences and side effects that each such request might have. A good rule is to use formatter requests only when absolutely necessary. The MM macros described herein should be used in most cases because:

- It is much easier to control (and change at any later point in time) overall style of the document.
- Complicated features (such as footnotes or tables of contents) can be obtained with ease.
- User is insulated from peculiarities of the formatter language.

### 3.2 Arguments and Double Quotes

For any macro call, a null argument is an argument whose width is zero. Such an argument often has a special meaning; the preferred form for a null argument is "". Omitting an argument is not the same as supplying a null argument (e.g., the .MT macro in {6.7}). Omitted arguments can occur only at the end of an argument list; null arguments can occur anywhere in the list.

Any macro argument containing ordinary (paddable) spaces must be enclosed in double quotes. A double quote (") is a single character that must not be confused with two apostrophes or acute accents (') or with two grave accents (`). Otherwise, it will be treated as several separate arguments.

Double quotes are not permitted as part of the value of a macro argument or of a string that is to be used as a macro argument. If it is necessary to have a macro argument value, two grave accents (`) and/or two acute accents (') may be used instead. This restriction is necessary because many macro arguments are processed (interpreted) a variable number of times. For example, headings are first printed in the text and may be reprinted in the table of contents.

### 3.3 Unpaddable Spaces

When output lines are justified to give an even right margin, existing spaces in a line may have additional spaces appended to them. This may distort the desired alignment of text. To avoid this distortion, it is necessary to specify a space that cannot be expanded during justification, i.e., an *unpaddable space*. There are several ways to accomplish this:

- The user may type a backslash followed by a space (\ ). This pair of characters directly generates an unpaddable space.
- The user may sacrifice some seldom-used character to be translated into a space upon output.

Because this translation occurs after justification, the chosen character may be used anywhere an unpaddable space is desired. The tilde (~) is often used with the translation macro for this purpose. To use the tilde in this way, the following is inserted at the beginning of the document:

```
.tr ~
```

If a tilde must actually appear in the output, it can be temporarily "recovered" by inserting

```
.tr ~ ~
```



before the place where needed. Its previous usage is restored by repeating the `.tr ~` after a break or after the line containing the tilde has been forced out.

**Note:** Use of the tilde in this fashion is not recommended for documents in which the tilde is used within equations.

### 3.4 Hyphenation

Formatters do not perform hyphenation unless requested. Hyphenation can be turned on in the body of the text by specifying

`.nr Hy 1`

once at the beginning of the document input file. Paragraph 8.3 describes hyphenation within footnotes and across pages,

If hyphenation is requested, formatters will automatically hyphenate words if need be. However, the user may specify hyphenation points for a specific occurrence of any word with a special character known as a hyphenation indicator or may specify hyphenation points for a small list of words (about 128 characters).

If the *hyphenation indicator* (initially, the 2-character sequence “\%”) appears at the beginning of a word, the word is not hyphenated. Alternatively, it can be used to indicate legal hyphenation points inside a word. All occurrences of the hyphenation indicator disappear on output.

The user may specify a different hyphenation indicator.

`.HC [hyphenation-indicator]`

The circumflex (^) is often used for this purpose by inserting the following at the beginning of a document input text file:

`.HC ^`

**Note:** Any word containing hyphens or dashes (also known as *em* dashes) will be hyphenated immediately after a hyphen or dash if it is necessary to hyphenate the word, even if the formatter hyphenation function is turned off.

The user may supply, via the exception word `.hw` request, a small list of words with the proper hyphenation points indicated. For example, to indicate the proper hyphenation of the word “printout”, the user may specify

`.hw print-out`

### 3.5 Tabs

Macros `.MT {6.7}`, `.TC {10.1}`, and `.CS {10.2}` use the formatter tabs `.ta` request to set tab stops and then restore the default values of tab settings (every eight characters in the `nroff` formatter; every 1/2 inch in the `troff` formatter). Setting tabs to other than the default values is the user's responsibility.

Default tab setting values are 9, 17, 25, ..., 161 for a total of 20 tab stops. Values may be separated by commas, spaces, or any other non-numeric character. A user may set tab stops at any value desired. For example:

`.ta 9 17 25 33 41 49 57 ... 161`

A tab character is interpreted with respect to its position on the input line rather than its position on the output line. In general, tab characters should appear only on lines processed in no-fill (`.nf`) mode {3.1}.



The `tbl(1)` program {7.3} changes tab stops but does not restore default tab settings.

### 3.6 BEL Character

The nonprinting character BEL is used as a delimiter in many macros to compute the width of an argument or to delimit arbitrary text, e.g., in page headers and footers {9}, headings {4}, and lists {5}. Users who include BEL characters in their input text file (especially in arguments to macros) will receive mangled output.

### 3.7 Bullets

A bullet (●) is often obtained on a typewriter terminal by using an “o” overstruck by a “+”. For compatibility with the `troff` formatter, a bullet string is provided by MM with the following sequence:

```
\*(BU
```

The bullet list (`.BL`) macro {5.1.1.2} uses this string to generate automatically the bullets for bullet-listed items.

### 3.8 Dashes, Minus Signs, and Hyphens

The `troff` formatter has distinct graphics for a dash, a minus sign, and a hyphen; the `nroff` formatter does not.

- Users who intend to use the `nroff` formatter only may use the minus sign (–) for the minus, hyphen, and dash.
- Users who plan to use the `troff` formatter primarily should follow `troff` escape conventions.
- Users who plan to use both formatters must take care during input text file preparation. Unfortunately, these graphic characters cannot be represented in a way that is both compatible and convenient for both formatters.

The following approach is suggested:

| SYMBOL | ACTION                                                                                                                                                                                                                                                                                                                                        |
|--------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Dash   | Type <code>\*(EM</code> for each text dash for both <code>nroff</code> and <code>troff</code> formatters. This string generates an em dash in the <code>troff</code> formatter and two dashes (—) in the <code>nroff</code> formatter. Dash list ( <code>.DL</code> ) macros {5.2.1.3} automatically generate the em dash for each list item. |
| Hyphen | Type <code>-</code> and use as is for both formatters. The <code>nroff</code> formatter will print it as is, and the <code>troff</code> formatter will print <code>-</code> (a true hyphen).                                                                                                                                                  |
| Minus  | Type <code>\-</code> for a true minus sign regardless of formatter. The <code>nroff</code> formatter will effectively ignore the “\”; the <code>troff</code> formatter will print a true minus sign.                                                                                                                                          |

### 3.9 Trademark String

A trademark string `\*(Tm` is available with MM. This places the letters “TM” one-half line above the text that it follows. For example:

```
The
.I
User's Manual—UNIX
.R
```



```
\h'-1\'*(Tm
.I
Operating System
.R
is available from the library.
```

yields:

The User's Manual—UNIX™ Operating System is available from the library.

### 3.10 Use of Formatter Requests

Most formatter requests should not be used with MM because MM provides the corresponding formatting functions in a much more user-oriented and surprise-free fashion than do the basic formatter requests. However, some formatter requests are useful with MM, namely the following:

|     |                                   |
|-----|-----------------------------------|
| .af | Assign format                     |
| .br | Break                             |
| .ce | Center                            |
| .de | Define macro                      |
| .ds | Define string                     |
| .fi | Fill output lines                 |
| .hw | Exception word                    |
| .ls | Line spacing                      |
| .nf | No filling of output lines        |
| .nr | Define and set number register    |
| .nx | Go to next file (does not return) |
| .rm | Remove macro                      |
| .rr | Remove register                   |
| .rs | Restore spacing                   |
| .so | Switch to source file and return  |
| .sp | Space                             |
| .ta | Tab stop settings                 |
| .ti | Temporary indent                  |
| .tl | Title                             |
| .tr | Translate                         |
| !   | Escape                            |

The .fp, .lg, and .ss requests are also sometimes useful for the troff formatter. Use of other requests without fully understanding their implications very often leads to disaster.

## 4. Paragraphs and Headings

### 4.1 Paragraphs

```
.P [type]
one or more lines of text.
```

The .P macro is used to control paragraph style.

#### 4.1.1 Paragraph Indentation

An indented or a nonindented paragraph is defined with the *type* argument.



| <i>type</i> | <i>Result</i>  |
|-------------|----------------|
| 0           | left justified |
| 1           | indent         |

In a left-justified paragraph, the first line begins at the left margin. In an indented paragraph, the paragraph is indented the amount specified in the *Pi* register (default value is 5). For example, to indent paragraphs by ten spaces, the following is entered at the beginning of the document input file:

```
.nr Pi 10
```

A document input file possesses a default paragraph type obtained by specifying ".P" before each paragraph that does not follow a heading {4.2}. Default paragraph type is controlled by the *Pt* number register. The initial value of *Pt* is 0, which provides left-justified paragraphs.

All paragraphs can be forced to be indented by inserting the following at the beginning of the document input file:

```
.nr Pt 1
```

All paragraphs can be indented except after headings, lists, and displays by entering the following at the beginning of the document input file:

```
.nr Pt 2
```

Both the *Pi* and *Pt* register values must be greater than zero for any paragraphs to be indented.

**Note:** Values that specify indentation must be unscaled and are treated as character positions, i.e., as a number of ens. In the *nroff* formatter, an en is equal to the width of a character. In the *troff* formatter, an en is the number of points (1 point = 1/72 of an inch) equal to half the current point size.

Regardless of the value of *Pt*, an individual paragraph can be forced to be left-justified or indented. The ".P 0" macro request forces left justification; ".P 1" causes indentation by the amount specified by the register *Pi*.

If .P occurs inside a list, the indent (if any) of the paragraph is added to the current list indent {5}.

#### 4.1.2 Numbered Paragraphs

Numbered paragraphs may be produced by setting the *Np* register to 1. This produces paragraphs numbered within first level headings, e.g., 1.01, 1.02, 1.03, 2.01, etc.

A different style of numbered paragraphs is obtained by using the *.nP* macro rather than the *.P* macro for paragraphs. This produces paragraphs that are numbered within second level headings.

```
.H 1 " FIRST HEADING "
.H 2 " Second Heading "
.nP
one or more lines of text
```

The paragraphs contain a "double-line indent" in which the text of the second line is indented to be aligned with the text of the first line so that the number stands out.

#### 4.1.3 Spacing Between Paragraphs

The *Ps* number register controls the amount of spacing between paragraphs. By default, *Ps* is set to 1, yielding one blank space (one-half a vertical space).



## 4.2 Numbered Headings

.H level [heading-text] [heading-suffix]  
zero or more lines of text

The .H macro provides seven levels of numbered headings. Level 1 is the highest; level 7 the lowest.

The *heading-suffix* argument is appended to the *heading-text* argument and may be used for footnote marks which should not appear with heading text in the table of contents.

**Note:** There is no need for a .P macro immediately after a .H or .HU {4.3} because the .H macro also performs the function of the .P macro. Any immediately following .P macro is ignored. It is, however, good practice to start every paragraph with a .P macro, thereby ensuring that all paragraphs uniformly begin with a .P throughout an entire document.

### 4.2.1 Normal Appearance

The effect of the .H macro varies according to argument level. First-level headings are preceded by two blank lines (one vertical space); all others are preceded by one blank line (one-half a vertical space). The following table describes the default effect of the level argument.

|                          |                                                                                                                                                                                                                                                                        |
|--------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| .H 1 heading-text        | Produces a bold font heading followed by a single blank line (one-half a vertical space). The following text begins on a new line and is indented according to the current paragraph type. Full capital letters should normally be used to make the heading stand out. |
| .H 2 heading-text        | Produces a bold font heading followed by a single blank line (one-half a vertical space). The following text begins on a new line and is indented according to the current paragraph type. Normally, initial capitals are used.                                        |
| .H <i>n</i> heading-text | Produces an underlined (italicized) heading followed by two spaces ( $3 < n < 7$ ). <i>The following text begins on the same line, i.e., these are run-in headings.</i>                                                                                                |

Appropriate numbering and spacing (horizontal and vertical) occur even if the heading-text argument is omitted from a .H macro call.

The following listing gives the first few .H calls used for this part.

```
.H 1 " paragraphs and headings "  
.H 2 " Paragraphs "  
.H 3 " Paragraph Indention "  
.H 3 " Numbered Paragraphs "  
.H 3 " Spacing Between Paragraphs "  
.H 2 " Numbered Headings "  
.H 3 " Normal Appearance "  
.H 3 " Altering Appearance "  
.H 4 " Prespacing and Page Ejection "  
.H 4 " Spacing After Headings "  
.H 4 " Centered Headings "
```



.H 4 " Bold, Italic, and Underlined Headings "

.H 5 "Control by Level"

**Note:** Users satisfied with the default appearance of headings may skip to the paragraph entitled "Un-numbered Headings" {4.3}.

#### 4.2.2 Altering Appearance

The user can modify the appearance of headings quite easily by setting certain registers and strings at the beginning of the document input text file. This permits quick alteration of a document's style because this style-control information is concentrated in a few lines rather than being distributed throughout the document.

##### 4.2.2.1 Prespacing and Page Ejection

A first-level heading (.H 1) normally has two blank lines (one vertical space) preceding it, and all other headings are preceded by one blank line (one-half a vertical space). If a multiline heading were to be split across pages, it is automatically moved to the top of the next page. Every first-level heading may be forced to the top of a new page by inserting

```
.nr Ej 1
```

at the beginning of the document input text file. Long documents may be made more manageable if each section starts on a new page. Setting the *Ej* register to a higher value causes the same effect for headings up to that level, i.e., a page eject occurs if the heading level is less than or equal to the *Ej* value.

##### 4.2.2.2 Spacing After Headings

Three registers control the appearance of text immediately following a .H call. The registers are *Hb* (heading break level), *Hs* (heading space level), and *Hi* (post-heading indent).

If the heading level is less than or equal to *Hb*, a break {3.1} occurs after the heading. If the heading level is less than or equal to *Hs*, a blank line (one-half a vertical space) is inserted after the heading. The default value for *Hb* and *Hs* is 2. If a heading level is greater than *Hb* and also greater than *Hs*, then the heading (if any) is run into the following text. These registers permit headings to be separated from the text in a consistent way throughout a document while allowing easy alteration of white space and heading emphasis.

For any stand-alone heading, i.e., a heading not run into the following text, alignment of the next line of output is controlled by the *Hi* register.

- If *Hi* is 0, text is left-justified.
- If *Hi* is 1 (the default value), text is indented according to the paragraph type as specified by the *Pt* register {4.1}.
- If *Hi* is 2, text is indented to line up with the first word of the heading itself so that the heading number stands out more clearly.

To cause a blank line (one-half a vertical space) to appear after the first three heading levels, to have no run-in headings, and to force the text following all headings to be left-justified (regardless of the value of *Pt*), the following should appear at the beginning of the document input text file:

```
.nr Hs 3  
.nr Hb 7  
.nr Hi 0
```



#### 4.2.2.3 Centered Headings

The *Hc* register can be used to obtain centered headings. A heading is centered if its level argument is less than or equal to *Hc* and if it is also a stand-alone heading {4.2.2.2}. The *Hc* register is 0 initially (no centered headings).

#### 4.2.2.4 Bold, Italic, and Underlined Headings

**4.2.2.4.1 Control by Level:** Any heading that is underlined by the **nroff** formatter is italicized by the **troff** formatter. The string *HF* (heading font) contains seven codes that specify fonts for heading levels 1 through 7. Legal codes, code interpretations, and defaults for *HF* codes are:

| FORMATTER    | HF CODE      |           |      | DEFAULT<br>HF CODE |
|--------------|--------------|-----------|------|--------------------|
|              | 1            | 2         | 3    |                    |
| <b>nroff</b> | no underline | underline | bold | 3 3 2 2 2 2 2      |
| <b>troff</b> | Roman        | italic    | bold | 3 3 2 2 2 2 2      |

Thus, levels 1 and 2 are bold; levels 3 through 7 are underlined by the **nroff** formatter and italicized by the **troff** formatter. The user may reset *HF* as desired. Any value omitted from the right end of the list is assumed to be a 1. The following request would result in five bold levels and two Roman font levels:

```
.ds HF 3 3 3 3 3
```

**4.2.2.4.2 NROFF Underlining Style:** The **nroff** formatter underlines in either of two styles:

- The normal style (**.ul** request) is to underline only letters and digits.
- The continuous style (**.cu** request) underlines all characters including spaces.

By default, MM attempts to use the continuous style on any heading that is to be underlined and is short enough to fit on a single line. If a heading is to be underlined but is longer than a single line, the heading is underlined in the normal style.

All underlining of headings can be forced to the normal style by using the **-rU1** flag when invoking the **nroff** formatter {2.4}.

**4.2.2.4.3 Heading Point Sizes:** The user may specify the desired point size for each heading level with the *HP* string (for use with the **troff** formatter only).

```
.ds HP [ps1] [ps2] [ps3] [ps4] [ps5] [ps6] [ps7]
```

By default, the text of headings (**.H** and **.HU**) is printed in the same point size as the body except that bold stand-alone headings are printed in a size one point smaller than the body. The string *HP*, similar to the string *HF*, can be specified to contain up to seven values, corresponding to the seven levels of headings. For example:

```
.ds HP 12 12 10 10 10 10 10
```

specifies that the first and second level headings are to be printed in 12-point type with the remainder printed



in 10-point. Specified values may also be relative point-size changes, for example:

.ds HP +2 +2 -1 -1

If absolute point sizes are specified, then absolute sizes will be used regardless of the point size of the body of the document. If relative point sizes are specified, then point sizes for headings will be relative to the point size of the body even if the latter is changed.

Null or zero values imply that default size will be used for the corresponding heading level.

**Note:** Only the point size of the headings is affected. Specifying a large point size without providing increased vertical spacing (via .HX and/or .HZ) may cause overprinting.

#### 4.2.2.5 Marking Styles—Numerals and Concatenation

.HM [arg1] ... [arg7]

The registers named *H1* through *H7* are used as counters for the seven levels of headings. Register values are normally printed using Arabic numerals. The .HM macro (heading mark style) allows this choice to be overridden thus providing "outline" and other document styles. This macro can have up to seven arguments; each argument is a string indicating the type of marking to be used. Legal arguments and their meanings are:

| ARGUMENT | MEANING                                                                 |
|----------|-------------------------------------------------------------------------|
| 1        | Arabic (default for all levels)                                         |
| 0001     | Arabic with enough leading zeroes to get the specified number of digits |
| A        | Uppercase alphabetic                                                    |
| a        | Lowercase alphabetic                                                    |
| I        | Uppercase Roman                                                         |
| i        | Lowercase Roman                                                         |

Omitted arguments are interpreted as 1; illegal arguments have no effect.

By default, the complete heading mark for a given level is built by concatenating the mark for that level to the right of all marks for all levels of higher value. To inhibit the concatenation of heading level marks, i.e., to obtain just the current level mark followed by a period, the heading mark type register (*Ht*) is set to 1. For example, a commonly used "outline" style is obtained by:

```
.HM I A 1 a i
.nr Ht 1
```

#### 4.3 Unnumbered Headings

.HU heading-text

The .HU macro is a special case of .H; it is handled in the same way as .H except that no heading mark is printed. In order to preserve the hierarchical structure of headings when .H and .HU calls are intermixed, each .HU heading is considered to exist at the level given by register *Hu*, whose initial value is 2. Thus, in the normal case, the only difference between

.HU heading-text



and

.H 2 heading-text

is the printing of the heading mark for the latter. Both macros have the effect of incrementing the numbering counter for level 2 and resetting to zero the counters for levels 3 through 7. Typically, the value of *Hu* should be set to make unnumbered headings (if any) be the lowest-level headings in a document.

The .HU macro can be especially helpful in setting up appendices and other sections that may not fit well into the numbering scheme of the main body of a document {14.2.1}.

#### 4.4 Headings and Table of Contents

The text of headings and their corresponding page numbers can be automatically collected for a table of contents. This is accomplished by doing the following:

- specifying in the contents level register, *Cl*, what level headings are to be saved
- invoking the .TC macro {10.1} at the end of the document.

Any heading whose level is less than or equal to the value of the *Cl* register is saved and later displayed in the table of contents. The default value for the *Cl* register is 2, i.e., the first two levels of headings are saved.

Due to the way headings are saved, it is possible to exceed the formatter's storage capacity, particularly when saving many levels of many headings, while also processing displays {7} and footnotes {8}. If this happens, the "Out of temp file space" formatter error message (Table 4.D) will be issued; the only remedy is to save fewer levels and/or to have fewer words in the heading text.

#### 4.5 First-Level Headings and Page Numbering Style

By default, pages are numbered sequentially at the top of the page. For large documents, it may be desirable to use page numbering of the "section-page" form where "section" is the number of the current first-level heading. This page numbering style can be achieved by specifying the -rN3 or -rN5 flag on the command line {9.3}. As a side effect, this also has the effect of setting *Ej* to 1, i.e., each first level section begins on a new page. In this style, the page number is printed at the bottom of the page so that the correct section number is printed.

#### 4.6 User Exit Macros

**Note:** This paragraph is intended primarily for users who are accustomed to writing formatter macros.

.HX dlevel rlevel heading-text  
.HY dlevel rlevel heading-text  
.HZ dlevel rlevel heading-text

The .HX, .HY, and .HZ macros are the means by which the user obtains a final level of control over the previously described heading mechanism. These macros are not defined by MM. These macros are intended to be defined by the user. The .H macro invokes .HX shortly before the actual heading text is printed; it calls .HZ as its last action. After .HX is invoked, the size of the heading is calculated. This processing causes certain features that may have been included in .HX, such as .ti for temporary indent, to be lost. After the size calculation, .HY is invoked so that the user may respecify these features. All default actions occur if these macros are not defined. If .HX, .HY, or .HZ are defined by the user, user-supplied definition is interpreted at the appropriate point. These macros can therefore influence handling of all headings because the .HU macro is actually a special case of the .H macro.

If the user originally invoked the .H macro, then the derived level (*dlevel*) and the real level (*rlevel*) are both equal to the level given in the .H invocation. If the user originally invoked the .HU macro {4.3}, *dlevel* is equal to the contents of register *Hu*, and *rlevel* is 0. In both cases, *heading-text* is the text of the original invocation.



By the time .H calls .HX, it has already incremented the heading counter of the specified level {4.2.2.5}, produced blank lines (vertical spaces) to precede the heading {4.2.2.1}, and accumulated the "heading mark", i.e., the string of digits, letters, and periods needed for a numbered heading. When .HX is called, all user-accessible registers and strings can be referenced, as well as the following:

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| string }0   | If <i>rlevel</i> is nonzero, this string contains the "heading mark". Two unpaddable spaces (to separate the <i>mark</i> from the <i>heading</i> ) have been appended to this string. If <i>rlevel</i> is 0, this string is null.                                                                                                                                                                                                                          |
| register ;0 | This register indicates the type of spacing that is to follow the heading {4.2.2.2}. A value of 0 means that the heading is run-in. A value of 1 means a break (but no blank line) is to follow the heading. A value of 2 means that a blank line (one-half a vertical space) is to follow the heading.                                                                                                                                                    |
| string }2   | If "register ;0" is 0, this string contains two unpaddable spaces that will be used to separate the (run-in) heading from the following text. If "register ;0" is nonzero, this string is null.                                                                                                                                                                                                                                                            |
| register ;3 | This register contains an adjustment factor for a .ne request issued before the heading is actually printed. On entry to .HX, it has the value 3 if <i>dlevel</i> equals 1, and 1 otherwise. The .ne request is for the following number of lines: the contents of the "register ;0" taken as blank lines (halves of vertical space) plus the contents of "register ;3" as blank lines (halves of vertical space) plus the number of lines of the heading. |

The user may alter the values of {0, }2, and ;3 within .HX. The following are examples of actions that might be performed by defining .HX to include the lines shown:

- Change first-level heading mark from format *n.* to *n.0*:  

```
.if \ $1=1 .ds }0\ n(H1.0\ <sp>\ <sp>
(where <sp> stands for a space)
```
- Separate run-in heading from the text with a period and two unpaddable spaces:  

```
.if \ n(;0=0 .ds } 2 .\ <sp>\ <sp>
```
- Assure that at least 15 lines are left on the page before printing a first-level heading:  

```
.if \ $1=1 .nr ;3 15- \ n(;0
```
- Add three additional blank lines before each first-level heading:  

```
.if \ $1=1 .sp 3
```
- Indent level 3 run-in headings by five spaces:  

```
.if \ $1=3 .ti 5n
```

If temporary strings or macros are used within .HX, their names should be chosen with care {14.1}.

When the .HY macro is called after the .ne is issued, certain features requested in .HX must be repeated. For example:

```
.de HY
.if \ $1=3 .ti 5n
..
```



The .HZ macro is called at the end of .H to permit user-controlled actions after the heading is produced. In a large document, sections may correspond to chapters of a book; and the user may want to change a page header or footer, e.g.:

```
.de HZ
.if \\$1=1 .PF " Section \\$3 "
..
```

#### 4.7 Hints for Large Documents

A large document is often organized for convenience into one input text file per section. If the files are numbered, it is wise to use enough digits in the names of these files for the maximum number of sections, i.e., use suffix numbers 01 through 20 rather than 1 through 9 and 10 through 20.

Users often want to format individual sections of long documents. To do this with the correct section numbers, it is necessary to set register *H1* to one less than the number of the section just before the corresponding .H\ 1 call. For example, at the beginning of Part 5, insert

```
.nr H1 4
```

**Note:** This is not good practice. It defeats the automatic (re)numbering of sections when sections are added or deleted. Such lines should be removed as soon as possible.

### 5. Lists

This part describes different styles of lists; automatically numbered and alphabetized lists, bullet lists, dash lists, lists with arbitrary marks, and lists starting with arbitrary strings, i.e., with terms or phrases to be defined.

#### 5.1 List Macros

In order to avoid repetitive typing of arguments to describe the style or appearance of items in a list, MM provides a convenient way to specify lists. All lists share the same overall structure and are composed of the following basic parts:

- A *list-initialization macro* (.AL .BL, .DL, .ML, .RL, or .VL) determines the style of list: line spacing, indentation, marking with special symbols, and numbering or alphabetizing of list items.
- One or more *list-item macros* (.LI) identifies each unique item to the system. It is followed by the actual text of the corresponding list item.
- The *list-end macro* (.LE) identifies the end of the list. It terminates the list and restores the previous indentation.

Lists may be nested up to six levels. The list-initialization macro saves the previous list status (indentation marking style, etc.); the .LE macro restores it.

With this approach, the format of a list is specified only once at the beginning of the list. In addition by building onto the existing structure, users may create their own customized sets of list macros with relatively little effort ({5.3} and {5.4}).

##### 5.1.1 List-Initialization Macros

List-initialization macros are implemented as calls to the more basic .LB macro {5.2}.



They are:

|     |                                             |
|-----|---------------------------------------------|
| .AL | Automatically Numbered or Alphabetized List |
| .BL | Bullet List                                 |
| .DL | Dash List                                   |
| .ML | Marked List                                 |
| .RL | Reference List                              |
| .VL | Variable-Item List                          |

#### 5.1.1.1 Automatically Numbered or Alphabetized List

`.AL [type] [text-indent] [1]`

The `.AL` macro is used to begin sequentially numbered or alphabetized lists. If there are no arguments, the list is numbered; and text is indented by *Li* (initially six) spaces from the indent in force when the `.AL` is called. This leaves room for a space, two digits, a period, and two spaces before the text. Values that specify indentation must be unscaled and are treated as "character positions", i.e., number of ens.

Spacing at the beginning of the list and between items can be suppressed by setting the list space register (*Ls*). The *Ls* register is set to the innermost list level for which spacing is done. For example:

`.nr Ls 0`

specifies that no spacing will occur around any list items. The default value for *Ls* is six (which is the maximum list nesting level).

- The *type* argument may be given to obtain a different type of sequencing. Its value indicates the first element in the sequence desired. If *type* argument is omitted or null, the value 1 is assumed.

| ARGUMENT | INTERPRETATION                  |
|----------|---------------------------------|
| 1        | Arabic (default for all levels) |
| A        | Uppercase alphabetic            |
| a        | Lowercase alphabetic            |
| I        | Uppercase Roman                 |
| i        | Lowercase Roman                 |

- If *text-indent* argument is non-null, it is used as the number of spaces from the current indent to the text, i.e., it is used instead of *Li* for this list only. If *text-indent* argument is null, the value of *Li* will be used.
- If the third argument is given, a blank line (one-half a vertical space) will not separate items in the list. A blank line will occur before the first item however.

#### 5.1.1.2 Bullet List

`.BL [text-indent] [1]`

The `.BL` macro begins a bullet list. Each list item is marked by a bullet (•) followed by one space.

- If the *text-indent* argument is non-null, it overrides the default indentation (the amount of paragraph indentation as given in the *Pi* register {4.1}). In the default case, the text of bullet and dash lists lines up with the first line of indented paragraphs.



- If the second argument is specified, no blank lines will separate items in the list.

#### 5.1.1.3 Dash List

.DL [text-indent] [1]

The .DL macro is identical to .BL except that a dash is used as the list item mark instead of a bullet.

#### 5.1.1.4 Marked List

.ML mark [text-indent] [1]

The .ML macro is much like .BL and .DL macros but expects the user to specify an arbitrary *mark* which may consist of more than a single character.

- Text is indented *text-indent* spaces if the second argument is not null; otherwise, the text is indented one more space than the width of *mark*.
- If the third argument is specified, no blank lines will separate items in the list.

**Note:** The *mark* must not contain ordinary (paddable) spaces because alignment of items will be lost if the right margin is justified {3.3}.

#### 5.1.1.5 Reference List

.RL [text-indent] [1]

A .RL macro call begins an automatically numbered list in which the numbers are enclosed by square brackets ([ ]).

- If the *text-indent* argument is non-null, it is used as the number of spaces from the current indent to the text, i.e., it is used instead of *Li* for this list only. If the *text-indent* argument is omitted or null, the value of *Li* is used.
- If the second argument is specified, no blank lines will separate the items in the list.

#### 5.1.1.6 Variable-Item List

.VL text-indent [mark-indent] [1]

When a list begins with a .VL macro, there is effectively no current *mark*; it is expected that each .LI will provide its own mark. This form is typically used to display definitions of terms or phrases.

- *Text-indent* provides the distance from current indent to beginning of the text.
- *Mark indent* produces the number of spaces from current indent to beginning of the *mark*, and it defaults to 0 if omitted or null.
- If the third argument is specified, no blank lines will separate items in the list.

An example of .VL macro usage is shown below:

```
.tr ~  
.VL 20 2
```



.LI mark~1

Here is a description of mark 1;  
 "mark 1" of the .LI line contains a tilde  
 translated to an unpaddable space in order  
 to avoid extra spaces between  
 "mark" and "1" {3.3}.

.LI second~mark.

This is the second mark also using a tilde translated to an unpaddable space.

.LI third~mark~longer~than~indent:

This item shows the effect of a long mark; one space separates the mark from the text.

.LI ~

This item effectively has no mark because the tilde following the .LI is translated into a space.

.LE

when formatted yields:

|                               |                                                                                                                                                                           |
|-------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| mark 1                        | Here is a description of mark 1; "mark 1" of the .LI line contains a tilde translated to an unpaddable space in order to avoid extra spaces between "mark" and "1" {3.3}. |
| second mark                   | This is the second mark also using a tilde translated to an unpaddable space.                                                                                             |
| third mark longer than indent | This item shows the effect of a long mark; one space separates the mark from the text.                                                                                    |
|                               | This item effectively has no mark because the tilde following the .LI is translated into a space.                                                                         |

The tilde argument on the last .LI above is required; otherwise, a "hanging indent" would have been produced. A "hanging indent" is produced by using .VL and calling .LI with no arguments or with a null first argument. For example:

.VL 10

.LI

Here is some text to show a hanging indent.

The first line of text is at the left margin.

The second is indented 10 spaces.

.LE

when formatted yields:

Here is some text to show a hanging indent. The first line of text is at the left margin. The second is indented 10 spaces.

**Note:** The *mark* must not contain ordinary (paddable) spaces because alignment of items will be lost if the right margin is justified {3.3}.

#### 5.1.2 List-Item Macro

.LI [mark] [1]

one or more lines of text that make up the list item.

The .LI macro is used with all lists and for each list item. It normally causes output of a single blank line (one-half a vertical space) before its list item although this may be suppressed.

- If no arguments are given, .LI labels the item with the current *mark* which is specified by the most recent list-initialization macro.



- If a single argument is given, that argument is output instead of the current *mark*.
- If two arguments are given, the first argument becomes a prefix to the current *mark* thus allowing the user to emphasize one or more items in a list. One unpaddable space is inserted between the prefix and the mark.

For example:

```
.BL 6
.LI
This is a simple bullet item.
.LI +
This replaces the bullet with a "plus".
.LI + 1
This uses a "plus" as prefix to the bullet.
.LE
```

when formatted yields:

- This is a simple bullet item.
- + This replaces the bullet with a "plus".
- + • This uses "plus" as prefix to the bullet.

**Note:** The *mark* must not contain ordinary (paddable) spaces because alignment of items will be lost if the right margin is justified {3.3}.

If the current *mark* (in the current list) is a null string and the first argument of *.LI* is omitted or null, the resulting effect is that of a "hanging indent", i.e., the first line of the following text is moved to the left starting at the same place where *mark* would have started {5.1.1.6}.

### 5.1.3 List-End Macro

```
.LE [1]
```

The *.LE* macro restores the state of the list to that existing just before the most recent list-initialization macro call. If the optional argument is given, the *.LE* outputs a blank line (one-half a vertical space). This option should generally be used only when the *.LE* is followed by running text but not when followed by a macro that produces blank lines of its own such as the *.P*, *.H*, or *.LI* macro.

The *.H* and *.HU* macros automatically clear all list information. The user may omit the *.LE* macros that would normally occur just before either of these macros and not receive the "LE:mismatched" error message. Such a practice is not recommended because errors will occur if the list text is separated from the heading at some later time (e.g., by insertion of text).

### 5.1.4 Example of Nested Lists

An example of input for the several lists and the corresponding output is shown below. The *.AL* and *.DL* macro calls {5.1.1} contained therein are examples of list-initialization macros. Input text is:

```
.AL A
.LI
This is alphabetized list item A.
```



This text shows the alignment of the second line of the item.

Notice the text indentations and alignment of left and right margins.

.AL

.LI

This is numbered item 1.

This text shows the alignment of the second line of the item.

The quick brown fox jumped over the lazy dog's back.

.DL

.LI

This is a dash item.

This text shows the alignment of the second line of the item.

The quick brown fox jumped over the lazy dog's back.

.LI + 1

This is a dash item with a "plus" as prefix.

This text shows the alignment of the second line of the item.

The quick brown fox jumped over the lazy dog's back.

.LE

.LI

This is numbered item 2.

.LE

.LI

This is another alphabetized list item B.

This text shows the alignment of the second line of the item.

The quick brown fox jumped over the lazy dog's back.

.LE

.P

This paragraph follows a list item and is aligned with the left margin.

A paragraph following a list resumes the normal line length and margins.



The output is:

- A. This is alphabetized list item A. This text shows the alignment of the second line of the item. Notice the text indentions and alignment of left and right margins.
1. This is numbered item 1. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.
    - This is a dash item. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.
    - + - This is a dash item with a "plus" as prefix. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.
  2. This is numbered item 2.
- B. This is another alphabetized list item B. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.

This paragraph follows a list item and is aligned with the left margin. A paragraph following a list resumes the normal line length and margins.

## 5.2 List-Begin Macro and Customized Lists

.LB text-indent mark-indent pad type [mark] [LI-space]\[LB-space]

List-initialization macros described above suffice for almost all cases. However, if necessary, the user may obtain more control over the layout of lists by using the basic list-begin macro (.LB). The .LB macro is used by the other list-initialization macros. Its arguments are as follows:

- The *text-indent* argument provides the number of spaces that text is to be indented from the current indent. Normally, this value is taken from the *Li* register (for automatic lists) or from the *Pi* register (for bullet and dash lists).
- The combination of *mark-indent* and *pad* arguments determines the placement of the mark. The mark is placed within an area (called *mark area*) that starts *mark-indent* spaces to the right of the current indent and ends where the text begins (i.e., ends *text-indent* spaces to the right of the current indent). The *mark-indent* argument is typically 0.
- Within the *mark area*, the mark is left justified if the *pad* argument is 0. If *pad* is a number *n* (greater than 0) then *n* blanks are appended to the mark; the *mark-indent* value is ignored. The resulting string immediately precedes the text. The *mark* is effectively right justified *pad* spaces immediately to the left of text.
- Arguments *type* and *mark* interact to control the type of marking used. If *type* is 0, simple marking is performed using the mark character(s) found in the *mark* argument. If *type* is greater than 0, automatic numbering or alphabetizing is done; and *mark* is then interpreted as the first item in the sequence to be used for numbering or alphabetizing, i.e., it is chosen from the set (1, A, a, I, i) as in {5.1.1.1}. This is summarized in the following table.



| <i>type</i> | <i>mark</i>              | <i>result</i>                                   |
|-------------|--------------------------|-------------------------------------------------|
| 0           | omitted                  | hanging indent                                  |
| 0           | <i>string</i>            | <i>string</i> is the mark                       |
| >0          | omitted                  | Arabic numbering                                |
| >0          | one of:<br>1, A, a, I, i | automatic numbering or<br>alphabetic sequencing |

Each nonzero value of *type* from one to six selects a different way of displaying the marks. The following table shows the output appearance for each value of *type*:

| VALUE | APPEARANCE |
|-------|------------|
| 1     | x.         |
| 2     | (x)        |
| 3     | (x)        |
| 4     | [x]        |
| 5     | <x>        |
| 6     | {x}        |

where x is the generated number or letter.

**Note:** The *mark* must not contain ordinary (paddable) spaces because alignment of items will be lost if the right margin is justified {3.3}.

- The *LI-space* argument gives the number of blank lines (halves of a vertical space) that should be output by each *.LI* macro in the list. If omitted, *LI-space* defaults to 1; the value 0 can be used to obtain compact lists. If *LI-space* is greater than 0, the *.LI* macro issues a *.ne* request for two lines just before printing the mark.
- The *LB-space* argument is the number of blank lines (one-half a vertical space) to be output by *.LB* itself. If omitted *LB-space* defaults to 0.

There are three combinations of *LI-space* and *LB-space*:

- The normal case is to set *LI-space* to 1 and *LB-space* to 0 yielding one blank line before each item in the list; such a list is usually terminated with a *.LE 1* macro to end the list with a blank line.
- For a more compact list, *LI-space* is set to 0, *LB-space* is set to 1, and the *.LE 1* macro is used at the end of the list. The result is a list with one blank line before and after it.
- If both *LI-space* and *LB-space* are set to 0 and the *.LE* macro is used to end the list, a list without any blank lines will result.

Paragraph {5.3} shows how to build upon the supplied list of macros to obtain other kinds of lists.

### 5.3 User-Defined List Structures

**Note:** This part is intended only for users accustomed to writing formatter macros.

If a large document requires complex list structures, it is useful to define the appearance for each list level only once instead of having to define the appearance at the beginning of each list. This permits consistency of



style in a large document. A generalized list-initialization macro might be defined in such a way that what the macro does depends on the list-nesting level in effect at the time the macro is called. Levels 1 through 5 of the lists to be formatted may have the following appearance:

A.

[1]

•

a)

+

The following code defines a macro (.aL) that always begins a new list and determines the type of list according to the current list level. To understand it, the user should know that the number register :g is used by the MM list macros to determine the current list level; it is 0 if there is no currently active list. Each call to a list-initialization macro increments :g, and each .LE call decrements it.

```
\ " register g is used as a local temporary to save
\ " :g before it is changed below
.de aL
.nr g \n(:g
.if \ng=0 .AL A          \ " produces an A.
.if \ng=1 .LB \n(Li 0 1 4 \ " produces a [1]
.if \ng=2 .BL           \ " produces a bullet
.
.if \ng=3 .LB \n(Li 0 2 2 a \ " produces an a)
.if \ng=4 .ML +         \ " produces a +
..
```

This macro can be used (in conjunction with .LI and .LE) instead of .AL, .RL, .BL, .LB, and .ML. For example, the following input:

```
.aL
.LI
First line.
.aL
.LI
Second line.
.LE
.LI
Third line.
.LE
```

when formatted yields

A. First line.

[1] Second line.

B. Third line.



There is another approach to lists that is similar to the .H mechanism. List-initialization, as well as the .LI and the .LE macros, are all included in a single macro. That macro (defined as .bL below) requires an argument to tell it what level of item is required; it adjusts the list level by either beginning a new list or setting the list level back to a previous value, and then issues a .LI macro call to produce the item.

```
.de bL
.ie \n($ .nr g \\\$1          \" if there is an argument,
                              \" that is the level
.el .nr g \\\n(:g            \" if no argument, use current level
.if \\\ng-\\\n(:g>1 .)D         \" **ILLEGAL SKIPPING OF LEVEL
                              \" increasing level by more than 1
.if \\\ng>\\\n(:g \\.aL \\\ng-1    \" if g > :g, begin new list
.nr                               \" and reset g to current level
                              \" (.aL changes g)
.if \\\n(:g>\\\ng .LC \\\ng        \" if :g > g, prune back to
                              \" correct level
                              \" if :g = g, stay within
                              \" current list
.LI                             \" in all cases, get out an item
..
```

For .bL to work, the previous definition of the .aL macro must be changed to obtain the value of *g* from its argument rather than from :*g*. Invoking .bL without arguments causes it to stay at the current list level. The .LC (List Clear) macro removes list descriptions until the level is less than or equal to that of its argument. For example, the .H macro includes the call ".LC 0". If text is to be resumed at the end of a list, insert the call ".LC 0" to clear out the lists completely. The example below illustrates the relatively small amount of input needed by this approach. The input text

The quick brown fox jumped over the lazy dog's back.

.bL 1

First line.

.bL 2

Second line.

.bL 1

Third line.

.bL

Fourth line.

.LC 0

Fifth line.

when formatted yields

The quick brown fox jumped over the lazy dog's back.

A. First line.

[1] Second line.

B. Third line.

C. Fourth line.

Fifth line.



## 6. Memorandum and Released-Paper Style Documents

One use of MM is for the preparation of memoranda and released-paper documents (a documentation style used by Bell Laboratories, Inc.) which have special requirements for the first page and for the cover sheet. Data needed (title, author, date, case numbers, etc.) is entered the same for both styles; an argument to the .MT macro indicates which style is being used.

### 6.1 Sequence of Beginning Macros

Macros, if present, must be given in the following order:

```
.ND new-date
.TL [charging-case] [filing-case]
one or more lines of text
.AF [company-name]
.AU name [initials] [loc] [dept] [ext] [room] [arg] [arg]
.AT [title] ...
.TM [number] ...
.AS [arg] [indent]
one or more lines of text
.AE.
.NS [arg]
one or more lines of text
.NE
.OK [keyword] ...
.MT [type] [addressee]
```

The only required macros for a memorandum for file or a released-paper document are .TL, .AU, and .MT; all other macros (and their associated input lines) may be omitted if the features are not needed. Once .MT has been invoked, none of the above macros (except .NS and .NE) can be reinvoked because they are removed from the table of defined macros to save memory space.

If neither the memorandum nor released-paper document style is desired, the TL, AU, TM, AE, OK, MT, ND, and AF macros should be omitted from the input text. If these macros are omitted, the first page will have only the page header followed by the body of the document.

### 6.2 Title

```
.TL [charging-case] [filing-case]
one or more lines of title text
```

Arguments to the .TL macro are the charging-case number(s) and filing-case number(s).

- The *charging-case* argument is the case number to which time was charged for the development of the project described in the memorandum. Multiple charging - case numbers are entered as "subarguments" by separating each from the previous with a comma and a space and enclosing the entire argument within double quotes.
- The *filing-case* argument is a number under which the memorandum is to be filed. Multiple filing case members are entered similarly. For example:

```
.TL "1 2 3 4 5, 6 7 8 9 0" 9 8 7 6 5 4 3 2 1
Construction of a Table of all Even Prime numbers
```

The title of the memorandum or released-paper document follows the .TL macro and is processed in fill mode.



The .br request may be used to break the title into several lines as follows:

```
.TL 12345
First Title Line
.br
\!br
Second Title Line
```

On output, the title appears after the word "subject" in the memorandum style and is centered and bold in the released-paper document style.

If only a charging case number or only a filing case number is given, it will be separated from the title in the memorandum style by a dash and will appear on the same line as the title. If both case numbers are given and are the same, then "Charging and Filing Case" followed by the number will appear on a line following the title. If the two case numbers are different, separate lines for "Charging Case" and "File Case" will appear after the title.

### 6.3 Authors

```
.AU name [initials] [loc] [dept] [ext] [room] [arg] [arg]
.AT [title] ...
```

The .AU macro receives as arguments information that describes an author. If any argument contains blanks, that argument must be enclosed within double quotes. The first six arguments must appear in the order given. A separate .AU macro is required for each author.

The .AT macro is used to specify the author's title. Up to nine arguments may be given. Each will appear in the signature block for memorandum style {6.11} on a separate line following the signer's name. The .AT must immediately follow the .AU for the given author. For example:

```
.AU " J. J. Jones " JJJ PY 9876 5432 1Z-234
.AT Director " Materials Research Laboratory "
```

In the "from" portion in the memorandum style, the author's name is followed by location and department number on one line and by room number and extension number on the next. The "x" for the extension is added automatically. Printing of the location, department number, extension number, and room number may be suppressed on the first page of a memorandum by setting the register *Au* to 0; the default value for *Au* is 1. Arguments 7 through 9 of the .AU macro, if present, will follow this normal author information in the "from" portion, each on a separate line. These last three arguments may be used for organizational numbering schemes, etc. For example:

```
.AU " S. P. Lename " SPL IH 9988 7766 5H-444 9876-543210.01MF
```

The name, initials, location, and department are also used in the signature block. Author information in the "from" portion, as well as names and initials in the signature block will appear in the same order as the .AU macros.

Names of authors in the released-paper style are centered below the title. Following the name of the last author, "Bell Laboratories" and the location are centered. The paragraph on memorandum types {6.7} contains information regarding authors from different locations.

### 6.4 TM Numbers

```
.TM [number] ...
```

If the memorandum is a technical memorandum, the TM numbers are supplied via the .TM macro. Up to



nine numbers may be specified. For example:

```
.TM 7654321 77777777
```

This macro call is ignored in the released-paper and external-letter styles {6.7}.

### 6.5 Abstract

```
.AS [arg] [indent]  
text of abstract  
.AE
```

If a memorandum has an abstract, the input is identified with the .AS (abstract start) and .AE (abstract end) delimiters. Abstracts are printed on page 1 of a document and/or on its cover sheet. There are three styles of cover sheet:

- released paper
- technical memorandum
- memorandum for file {10.2} (also used for engineer's notes, memoranda for record, etc.).

Cover sheets for released papers and technical memoranda are obtained by invoking the .CS macro {10.2}.

In released-paper style (first argument of the .MT macro {6.7} is 4) and in technical memorandum style if the first argument of .AS is:

- 0 Abstract will be printed on page 1 and on the cover sheet (if any).
- 1 Abstract will appear only on the cover sheet (if any).

In memoranda for file style and in all other documents (other than external letters) if the first argument of .AS is:

- 0 Abstract will appear on page 1 and there will be no cover sheet printed.
- 2 Abstract will appear only on the cover sheet which will be produced automatically (i.e., without invoking the .CS macro).

It is not possible to get either an abstract or a cover sheet with an external letter (first argument of the .MT macro is 5).

Notations such as a "copy to" list {6.11} are allowed on memorandum for file cover sheets; the .NS and .NE macros must appear after the .AS 2 and .AE macros. Headings {4.2, 4.3} and displays {7} are not permitted within an abstract.

The abstract is printed with ordinary text margins; an indentation to be used for both margins can be specified as the second argument of .AS. Values that specify indentation must be unscaled and are treated as "character positions", i.e., as the number of *ens*.

### 6.6 Other Keywords

```
.OK [keyword] ...
```

Topical keywords should be specified on a technical memorandum cover sheet. Up to nine such keywords or keyword phrases may be specified as arguments to the .OK macro; if any keyword contains spaces, it must be enclosed within double quotes.



## 6.7 Memorandum Types

`.MT [type] [addressee]`

The `.MT` macro controls the format of the top part of the first page of a memorandum or of a released-paper document and the format of the cover sheets. The *type* arguments and corresponding values are:

| <i>type</i>            | <i>Value</i>                       |
|------------------------|------------------------------------|
| <code>""</code>        | no memorandum type printed         |
| <code>0</code>         | no memorandum type printed         |
| <code>none</code>      | MEMORANDUM FOR FILE                |
| <code>1</code>         | MEMORANDUM FOR FILE                |
| <code>2</code>         | PROGRAMMER'S NOTES                 |
| <code>3</code>         | ENGINEER'S NOTES                   |
| <code>4</code>         | released-paper style               |
| <code>5</code>         | external-letter style              |
| <code>" string"</code> | <i>string</i> (enclosed in quotes) |

If the *type* argument indicates a memorandum style document, the corresponding statement indicated under "Value" will be printed after the last line of author information. If *type* is longer than one character, then the string itself will be printed. For example:

`.MT " Technical Note #5"`

A simple letter is produced by calling `.MT` with a null (but not omitted) or 0 argument.

The second argument to `.MT` is the name of the addressee of a letter. If present, that name and the page number replace the normal page header on the second and following pages of a letter. For example:

`.MT 1 " John Jones"`

produces

John Jones — 2

The *addressee* argument may not be used if the first argument is 4 (released-paper style document).

The released-paper style is obtained by specifying

`.MT 4 [1]`

This results in a centered, bold title followed by centered names of authors. The location of the last author is used as the location following "Bell Laboratories" (unless the `.AF` macro specifies a different company). If the optional second argument to `.MT 4` is given, then the name of each author is followed by the respective company



name and location. Information necessary for the memorandum style document but not for the released-paper style document is ignored.

If the released-paper style document is utilized, most BTL location codes are defined as strings that are the addresses of the corresponding BTL locations. These codes are needed only until the .MT macro is invoked. Thus, following the .MT macro, the user may reuse these string names. In addition, the macros for the end of a memorandum {6.11} and their associated lines of input are ignored when the released-paper style is specified.

Authors from non-BTL locations may include their affiliations in the released-paper style by specifying the appropriate .AF macro {6.9} and defining a string (with a 2-character name such as ZZ) for the address before each .AU. For example:

```
.TL
A Learned Treatise
.AF "Getem Inc."
.ds ZZ "22 Maple Avenue, Sometown 09999"
.AU "F. Swatter" "" ZZ
.AF "Bell Laboratories"
.AU "Sam P. Lename" "" CB
.MT 4 1
```

In the external-letter style document (.MT 5), only the title (without the word "subject:") and the date are printed in the upper left and right corners, respectively, on the first page. It is expected that preprinted stationery will be used with the company logo and address of the author.

#### 6.8 Date Changes

.ND new-date

The .ND macro alters the value of the string *DT*, which is initially set to produce the current date. If the argument contains spaces, it must be enclosed within double quotes.

#### 6.9 Alternate First-Page Format

.AF [company-name]

An alternate first-page format can be specified with the .AF macro. The words "subject", "date", and "from" (in the memorandum style) are omitted and an alternate company name is used.

If an argument is given, it replaces "Bell Laboratories" without affecting other headings. If the argument is null, "Bell Laboratories" is suppressed; and extra blank lines are inserted to allow room for stamping the document with a Bell System logo or a Bell Laboratories stamp.

The .AF with no argument suppresses "Bell Laboratories" and the "Subject/Date/From" headings, thus allowing output on preprinted stationery. The use of .AF with no arguments is equivalent to the use of -rA1 {2.4}, except that the latter must be used if it is necessary to change the line length and/or page offset (which default to 5.8i and 1i, respectively, for preprinted forms). The command line options -rOk and -rWk {2.4} are not effective with .AF. The only .AF use appropriate for the troff formatter is to specify a replacement for "Bell Laboratories".

The command line option -rEn {2.4} controls the font of the "Subject/Date/From" block.

#### 6.10 Example

Input text for a document may begin as follows:

```
.TL
```



```

MM\*(EMMemorandum Macros
.AU "D. W. Smith" DWS PY
.AU "J. R. Mashey" JRM PY
.AU "E. C. Pariser (January 1980 Revision)" ECP PY
.AU "N. W. Smith (June 1980 Revision)" NWS PY
.MT 4

```

Figure 4.1 shows the input text file and both the **nroff** and **troff** formatter outputs for a simple letter.

### 6.11 End of Memorandum Macros

At the end of a memorandum document (but not of a released-paper document), signatures of authors and a list of notations can be requested. The following macros and their input are ignored if the released-paper style document is selected.

#### 6.11.1 Signature Block

```

.FC [closing]
.SG [arg] [1]

```

The **.FC** macro prints "Yours very truly," as a formal closing, if no argument is used. It must be given before the **.SG** macro. A different closing may be specified as an argument to **.FC**.

The **.SG** macro prints the author's name(s) after the formal closing, if any. Each name begins at the center of the page. Three blank lines are left above each name for the actual signature.

- If no arguments are given, the line of reference data (location code, department number, author's initials, and typist's initials, all separated by hyphens) will not appear.
- A non-null first argument is treated as the typist's initials and is appended to the reference data.
- A null first argument prints reference data without the typist's initials or the preceding hyphen.
- If there are several authors and if the second argument is given, reference data is placed on the line of the first author.

Reference data contains only the location and department number of the first author. Thus, if there are authors from different departments and/or from different locations, the reference data should be supplied manually after the invocation (without arguments) of the **.SG** macro. For example:

```

.SG
.rs
.sp -1v
PY/MH-9876/5432-JJJ/SPL-cen

```

#### 6.11.2 "Copy to" and Other Notations

```

.NS [arg]
zero or more lines of the notation
.NE

```

Many types of notations (such as a list of attachments or "Copy to" lists) may follow signature and reference data. Various notations are obtained through the **.NS** macro, which provides for proper spacing and for breaking notations across pages, if necessary.



Codes for *arg* and the corresponding notations are:

| <i>arg</i>        | Notations              |
|-------------------|------------------------|
| <i>none</i>       | Copy to                |
| " "               | Copy to                |
| 0                 | Copy to                |
| 1                 | Copy (with att.) to    |
| 2                 | Copy (without att.) to |
| 3                 | Att.                   |
| 4                 | Atts.                  |
| 5                 | Enc.                   |
| 6                 | Encs.                  |
| 7                 | Under Separate Cover   |
| 8                 | Letter to              |
| 9                 | Memorandum to          |
| " <i>string</i> " | Copy (string) to       |

If *arg* consists of more than one character, it is placed within parentheses between the words "Copy" and "to". For example:

.NS " with att. 1 only "

will generate

Copy (with att. 1 only) to

as the notation. More than one notation may be specified before the .NE macro because a .NS macro terminates the preceding notation, if any. For example:

```
.NS 4
Attachment 1-List of register names
Attachment 2-List of string and macro names
.NS 1
J. J. Jones
.NS 2
S. P. Lename
G. H. Hurtz
.NE
```

would be formatted as

```
Atts.
Attachment 1—List of register names
Attachment 2—List of string and macro names

Copy (with att.) to
J. J. Jones

Copy (without att.) to
S. P. Lename
G. H. Hurtz
```

The .NS and .NE macros may also be used at the beginning following .AS 2 and .AE to place the notation list on the memorandum for file cover sheet {6.5}. If notations are given at the beginning without .AS 2, they will be saved and output at the end of the document.



### 6.11.3 Approval Signature Line

.AV approver's-name

The .AV macro may be used after the last notation block to automatically generate a line with spaces for the approval signature and date. For example:

.AV " Jane Doe "

produces

APPROVED:

\_\_\_\_\_  
Jane Doe

\_\_\_\_\_  
Date

### 6.12 One-Page Letter

At times, the user may like more space on the page forcing the signature or items within notations to the bottom of the page so that the letter or memo is only one page in length. This can be accomplished by increasing the page length with the `-rLn` option, e.g., `-rL90`. This has the effect of making the formatter believe that the page is 90 lines long and therefore providing more space than usual to place the signature or the notations.

**Note:** This will work only for a single-page letter or memo.

## 7. Displays

Displays are blocks of text that are to be kept together on a page and not split across pages. They are processed in an environment that is different from the body of the text (see the `.ev` request). The MM package provides two styles of displays—a *static* (.DS) style and a *floating* (.DF) style.

- In the *static* style, the display appears in the same relative position in the output text as it does in the input text. This may result in extra white space at the bottom of the page if the display is too long to fit in the remaining page space.
- In the *floating* style, the display "floats" through the input text to the top of the next page if there is not enough space on the current page. Thus input text that follows a floating display may precede it in the output text. A queue of floating displays is maintained so that their relative order of appearance in the text is not disturbed.

By default, a display is processed in no-fill mode with single spacing and is not indented from the existing margins. The user can specify indentation or centering as well as fill-mode processing.

**Note:** Displays and footnotes {8} may never be nested in any combination. Although lists {5} and paragraphs {4.1} are permitted, no headings (.H or .HU) {4.2, 4.3} can occur within displays or footnotes.

### 7.1 Static Displays

```
.DS [format].[fill] [rindent]
one or more lines of text
.DE
```



A static display is started by the `.DS` macro and terminated by the `.DE` macro. With no arguments, `.DS` accepts lines of text exactly as typed (no-fill mode) and will not indent lines from the prevailing left margin indentation or from the right margin.

- The *format* argument is an integer or letter used to control the left margin indentation and centering with the following meanings:

| <i>format</i> | <i>Meaning</i>            |
|---------------|---------------------------|
| " "           | no indent                 |
| 0 or L        | no indent                 |
| 1 or I        | indent by standard amount |
| 2 or C        | center each line          |
| 3 or CB       | center as a block         |
| omitted       | no indent                 |

- The *fill* argument is an integer or letter and can have the following meanings:

| <i>fill</i> | <i>Meaning</i> |
|-------------|----------------|
| " "         | no-fill mode   |
| 0 or N      | no-fill mode   |
| 1 or F      | fill mode      |
| omitted     | no-fill mode   |

- The *rindent* argument is the number of characters that the line length should be decreased, i.e., an indentation from the right margin. This number must be unscaled in the `nroff` formatter and is treated as *ens*. It may be scaled in the `troff` formatter or else defaults to *ems*.

The standard amount of static display indentation is taken from the *Si* register, a default value of five spaces. Thus, text of an indented display aligns with the first line of indented paragraphs, whose indent is contained in the *Pi* register {4.1}. Even though their initial values are the same (default values), these two registers are independent.

The display *format* argument value 3 (or CB) centers (horizontally) the entire display as a block (as opposed to `.DS 2` and `.DF 2` which center each line individually). All collected lines are left justified, and the display is centered based on width of the longest line. This format must be used in order for the `eqn/neqn` "mark" and "lineup" feature to work with centered equations {7.4}.

By default, a blank line (one-half a vertical space) is placed before and after *static* and *floating* displays. These blank lines before and after *static* displays can be inhibited by setting the register *Ds* to 0.

The following example shows usage of all three arguments for *static* displays. This block of text will be indented five spaces from the left margin, filled, and indented five spaces from the right margin (i.e., centered). The input text

```
.DS I F 5
```

```
"We the people of the United States,
in order to form a more perfect union,
establish justice, ensure domestic tranquillity,
provide for the common defense,
and secure the blessings of liberty to
ourselves and our posterity,
do ordain and establish this Constitution for the
```



United States of America."  
 .DE

produces

"We the people of the United States, in order to form a more perfect union, establish justice, ensure domestic tranquillity, provide for the common defense, and secure the blessings of liberty to ourselves and our posterity, do ordain and establish this Constitution to the United States of America."

## 7.2 Floating Displays

```
.DF [format] [fill] [rindent]
one or more lines of text
.DE
```

A floating display is started by the **.DF** macro and terminated by the **.DE** macro. Arguments have the same meanings as static displays described above, except indent, no indent, and centering are calculated with respect to the initial left margin. This is because prevailing indent may change between the time the formatter first reads the floating display and when the display is printed. One blank line (one-half a vertical space) occurs before and after a floating display.

The user may exercise precise control over the output positioning of floating displays through the use of two number registers, *De* and *Df* (see below). When a floating display is encountered by the **nroff** or **troff** formatter, it is processed and placed onto a queue of displays waiting to be output. Displays are removed from the queue and printed in the order entered, which is the order they appeared in the input file. If a new floating display is encountered and the queue of displays is empty, the new display is a candidate for immediate output on the current page. Immediate output is governed by size of display and the setting of the *Df* register code. The *De* register code controls whether text will appear on the current page after a floating display has been produced.

As long as the display queue contains one or more displays, new displays will be automatically entered there, rather than being output. When a new page is started (or the top of the second column when in 2-column mode), the next display from the queue becomes a candidate for output if the *Df* register code has specified "top-of-page" output. When a display is output, it is also removed from the queue.

When the end of a section (using section-page numbering) or the end of a document is reached, all displays are automatically removed from the queue and output. This occurs before a **.SG**, **.CS**, or **.TC** macro is processed.

A display will fit on the current page if there is enough room to contain the entire display or if the display is longer than one page in length and less than half of the current page has been used. A wide (full-page width) display will not fit in the second column of a 2-column document.

The *De* and *Df* number register code settings and actions are as follows:

### *De* Register

CODE

ACTION

0

No special action occurs (also the default condition).



| CODE | ACTION                                                                                                                                              |
|------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| 1    | A page eject will always follow the output of each floating display, so only one floating display will appear on a page and no text will follow it. |

**Note:** For any other code, the action performed is the same as for code 1.

### **Df Register**

| CODE | ACTION                                                                                                               |
|------|----------------------------------------------------------------------------------------------------------------------|
| 0    | Floating displays will not be output until end of section (when section-page numbering) or end of document.          |
| 1    | Output new floating display on current page if there is space; otherwise, hold it until end of section or document.  |
| 2    | Output exactly one floating display from queue to the top of a new page or column (when in 2-column mode).           |
| 3    | Output one floating display on current page if there is space; otherwise, output to the top of a new page or column. |
| 4    | Output as many displays as will fit (at least one) starting at the top of a new page or column.                      |

**Note:** If *De* register is set to 1, each display will be followed by a page eject causing a new top of page to be reached where at least one more display will be output (this also applies to code 5).

|   |                                                                                                                                                                                                                 |
|---|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 5 | Output a new floating display on the current page if there is room (also the default condition).<br>Output as many displays (at least one) as will fit on the page starting at the top of a new page or column. |
|---|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

**Note:** For any code greater than 5, the action performed is the same as for code 5.

The .WC macro {12.4} may also be used to control handling of displays in double-column mode and to control the break in text before floating displays.

### **7.3 Tables**

```
.TS [H]
global options;
column descriptors.
title lines
[.TH [N]]
data within the table.
.TE
```

The .TS (table start) and .TE (table end) macros make possible the use of the `tbl(1)` program. These macros are used to delimit text to be examined by `tbl` and to set proper spacing around the table. The display function and the `tbl` delimiting function are independent. In order to permit the user to keep together blocks that contain any mixture of tables, equations, filled text, unfilled text, and caption lines, the .TS/.TE block should be enclosed within a display (.DS/.DE). Floating tables may be enclosed inside floating displays (.DF/.DE).

Macros .TS and .TE permit processing of tables that extend over several pages. If a table heading is needed for each page of a multipage table, the "H" argument should be specified to the .TS macro as above. Following



the options and format information, table title is typed on as many lines as required and is followed by the .TH macro. The .TH macro must occur when ".TS H" is used for a multipage table. This is not a feature of tbl but of the definitions provided by the MM macro package.

The .TH (table header) macro may take as an argument the letter N. This argument causes the table header to be printed only if it is the first table header on the page. This option is used when it is necessary to build long tables from smaller .TS H/.TE segments. For example:

```
.TS H
global options;
column descriptors.
Title lines
.TH
data
.TE
.TS H
global options;
column descriptors.
Title lines
.TH N
data
.TE
```

will cause the table heading to appear at the top of the first table segment and no heading to appear at the top of the second segment when both appear on the same page. However, the heading will still appear at the top of each page that the table continues onto. This feature is used when a single table must be broken into segments because of table complexity (e.g., too many blocks of filled text). If each segment had its own .TS H/.TH sequence, it would have its own header. However, if each table segment after the first uses .TS H/.TH N, the table header will appear only at the beginning of the table and the top of each new page or column that the table continues onto.

For the **nroff** formatter, the **-e** option [**-E** for **mm(1)** {2.1}] may be used for terminals, such as the 450, that are capable of finer printing resolution. This will cause better alignment of features such as the lines forming the corner of a box. The **-e** is not effective with **col(1)**.

#### 7.4 Equations

```
.DS
.EQ [label]
equation(s)
.EN
.DE
```

Mathematical typesetting programs **eqn(1)** and **neqn** expect to use the .EQ (equation start) and .EN (equation end) macros as delimiters in the same way that **tbl(1)** uses .TS and .TE; however, .EQ and .EN must occur inside a .DS/.DE pair. There is an exception to this rule — if .EQ and .EN are used to specify only the delimiters for in-line equations or to specify **eqn/neqn** defines, the .DS and .DE macros must not be used; otherwise, extra blank lines will appear in the output.

The .EQ macro takes an argument that will be used as a label for the equation. By default, the label will appear at the right margin in the "vertical center" of the general equation. The *Eq* register may be set to 1 to change labeling to the left margin.

The equation will be centered for centered displays; otherwise, the equation will be adjusted to the opposite margin from the label.



### 7.5 Figure, Table, Equation, and Exhibit Titles

```
.FG [title] [override] [flag]  
.TB [title] [override] [flag]  
.EC [title] [override] [flag]  
.EX [title] [override] [flag]
```

The .FG (figure title), .TB (table title), .EC (equation caption), and .EX (exhibit caption) macros are normally used inside .DS/.DE pairs to automatically number and title figures, tables, and equations. These macros use registers *Fg*, *Tb*, *Ec*, and *Ex*, respectively (see paragraph {2.4} on -rN5 to reset counters in sections). For example:

```
.FG " This is a Figure Title "
```

yields

**Figure 1.** This is a Figure Title

The .TB macro replaces "Figure" with "TABLE", the .EC macro replaces "Figure" with "Equation", and the .EX macro replaces "Figure" with "Exhibit". The output title is centered if it can fit on a single line; otherwise, all lines but the first are indented to line up with the first character of the title. The format of the numbers may be changed using the .af request of the formatter. The format of the caption may be changed from

Figure 1. Title

to

Figure 1—Title

by setting the *Of* register to 1.

The *override* argument is used to modify normal numbering. If *flag* argument is omitted or 0, *override* is used as a prefix to the number; if the *flag* argument is 1, *override* is used as a suffix; and if the *flag* argument is 2, *override* replaces the number. If -rN5 {2.4} is given, "section-figure" numbering is set automatically and user-specified *override* string is ignored.

As a matter of formatting style, table headings are usually placed above the text of tables, while figure, equation, and exhibit titles are usually placed below corresponding figures and equations.

### 7.6 List of Figures, Tables, Equations, and Exhibits

A list of figures, tables, exhibits, and equations are printed following the table of contents if the number registers *Lf*, *Lt*, *Lx*, and *Le* (respectively) are set to 1. The *Lf*, *Lt*, and *Lx* registers are 1 by default; *Le* is 0 by default.

Titles of these lists may be changed by redefining the following strings which are shown here with their default values:

```
.ds Lf LIST OF FIGURES  
.ds Lt LIST OF TABLES  
.ds Lx LIST OF EXHIBITS  
.ds Le LIST OF EQUATIONS
```



## 8. Footnotes

There are two macros (.FS and .FE) that delimit text of footnotes, a string that automatically numbers footnotes, and a macro (.FD) that specifies the style of footnote text. Footnotes are processed in an environment different from that of the body of text. Refer to .ev request.

### 8.1 Automatic Numbering of Footnotes

Footnotes may be automatically numbered by typing the three characters “\\*F” (i.e., invoking the string *F*) immediately after the text to be footnoted without any intervening spaces. This will place the next sequential footnote number (in a smaller point size) a half line above the text to be footnoted.

### 8.2 Delimiting Footnote Text

```
.FS [label]
one or more lines of footnote text
.FE
```

There are two macros that delimit the text of each footnote. The .FS (footnote start) macro marks the beginning of footnote text, and the .FE (footnote end) macro marks the end. The *label* on the .FS, if present, will be used to mark footnote text. Otherwise, the number retrieved from the string *F* will be used. Automatically numbered and user-labeled footnotes may be intermixed. If a footnote is labeled (.FS *label*), the text to be footnoted must be followed by *label*, rather than by “\\*F”. Text between .FS and .FE is processed in fill mode. Another .FS, a .DS, or a .DF are not permitted between .FS and .FE macros. If footnotes are required in the title, abstract, or table {7.3} only labeled footnotes will appear properly. Everywhere else automatically numbered footnotes work correctly. For example:

#### *Automatically numbered footnote:*

```
This is the line containing the word \*F
.FS
This is the text of the footnote.
.FE
to be footnoted.
```

#### *Labeled footnote:*

```
This is a labeled*
.FS *

The footnote is labeled with an asterisk.
.FE
footnote.
```

Text of the footnote (enclosed within the .FS/.FE pair) should immediately follow the word to be footnoted in the input text, so that “\\*F” or *label* occurs at the end of a line of input and the next line is the .FS macro call. It is also good practice to append an unpaddable space {3.3} to “\\*F” or *label* when they follow an end-of-sentence punctuation mark (i.e., period, question mark, exclamation point).

Figure 4.2 illustrates the various available footnote styles as well as numbered and labeled footnotes.

### 8.3 Format Style of Footnote Text

```
.FD [arg] [1]
```



Within footnote text, the user can control formatting style by specifying text hyphenation, right margin justification, and text indentation, as well as left or right justification of the label when text indenting is used. The `.FD` macro is invoked to select the appropriate style.

The first argument is a number from the left column of the following table. Formatting style for each number is indicated in the remaining four columns. Further explanation of the first two of these columns is given in the definitions of the `.ad`, `.hy`, `.na`, and `.nh` (adjust, hyphenation, no adjust, and no hyphenation, respectively) requests in the `nroff` part of this document.

| <u>arg</u> | <u>HYPHENATION</u> | <u>ADJUST</u> | <u>TEXT<br/>INDENT</u> | <u>LABEL<br/>JUSTIFICATION</u> |
|------------|--------------------|---------------|------------------------|--------------------------------|
| 0          | .nh                | .ad           | yes                    | left                           |
| 1          | .hy                | .ad           | yes                    | left                           |
| 2          | .nh                | .na           | yes                    | left                           |
| 3          | .hy                | .na           | yes                    | left                           |
| 4          | .nh                | .ad           | no                     | left                           |
| 5          | .hy                | .ad           | no                     | left                           |
| 6          | .nh                | .na           | no                     | left                           |
| 7          | .hy                | .na           | no                     | left                           |
| 8          | .nh                | .ad           | yes                    | right                          |
| 9          | .hy                | .ad           | yes                    | right                          |
| 10         | .nh                | .na           | yes                    | right                          |
| 11         | .hy                | .na           | yes                    | right                          |

If the argument to `.FD` is greater than 11, the effect is as if `".FD 0"` were specified. If the first argument is omitted or null, the effect is equivalent to `".FD 10"` in the `nroff` formatter and to `".FD 0"` in the `troff` formatter; these are also the respective initial default values.

If the second argument is specified, then when a first-level heading is encountered, automatically numbered footnotes begin again with 1. This is most useful with the "section-page" page numbering scheme. As an example, the input line

```
.FD "" 1
```

maintains the default formatting style and causes footnotes to be numbered afresh after each first-level heading in a document.

Hyphenation across pages is inhibited by MM except for long footnotes that continue to the following page. If hyphenation is permitted, it is possible for the last word on the last line on the current page footnote to be hyphenated. The user has control over this situation by specifying an even `.FD` argument.

Footnotes are separated from the body of the text by a short line rule. Those that continue to the next page are separated from the body of the text by a full-width rule. In the `troff` formatter, footnotes are set in type two points smaller than the point size in the body of text.

#### 8.4 Spacing Between Footnote Entries

Normally, one blank line (a 3-point vertical space) separates footnotes when more than one occurs on a page. To change this spacing, the `Fs` number register is set to the desired value. For example:

```
.nr Fs 2
```

will cause two blank lines (a 6-point vertical space) to occur between footnotes.



## 9. Page Headers and Footers

Text printed at the top of each page is called *page header*. Text printed at the bottom of each page is called *page footer*. There can be up to three lines of text associated with the header — every page, even page only, and odd page only. Thus the page header may have up to two lines of text — the line that occurs at the top of every page and the line for the even- or odd-numbered page. The same is true for the page footer.

This part describes the default appearance of page headers and page footers and ways of changing them. The term *header* (not qualified by *even* or *odd*) is used to mean the page header line that occurs on every page, and similarly for the term *footer*.

### 9.1 Default Headers and Footers

By default, each page has a centered page number as the header. There is no default footer and no even/odd default headers or footers except as specified in paragraph {9.3}.

In a memorandum or a released-paper style document, the page header on the first page is automatically suppressed provided a break does not occur before the .MT macro is called. Macros and text in the following categories do not cause a break and are permitted before the memorandum types (.MT) macro:

- Memorandum and released-paper style document macros (.TL, .AU, .AT, .TM, .AS, .AE, .OK, .ND, .AF, .NS, and .NE)
- Page headers and footers macros (.PH, .EH, .OH, .PF, .EF, and .OF)
- The .nr and .ds requests.

### 9.2 Header and Footer Macros

For header and footer macros (.PH, .EH, .OH, .PF, .EF, and .OF), the argument [arg] is of the following form:

" 'left-part'center-part'right-part' "

If it is inconvenient to use apostrophe (') as the delimiter because it occurs within one of the parts, it may be replaced uniformly by any other character. In formatted output, the parts are left justified, centered, and right justified, respectively.

#### 9.2.1 Page Header

.PH [arg]

The .PH macro specifies the header that is to appear at the top of every page. The initial value is the default centered page number enclosed by hyphens. The page number contained in the *P* register is an Arabic number. The format of the number may be changed by the .af macro request.

If "*debug mode*" is set using the flag -rD1 on the command line {2.4}, additional information printed at the top left of each page is included in the default header. This consists of the Source Code Control System (SCCS) release and level of MM (thus identifying the current version {12.3}) followed by the current line number within the current input file.

#### 9.2.2 Even-Page Header

.EH [arg]



The .EH macro supplies a line to be printed at the top of each even-numbered page immediately following the header. Initial value is a blank line.

### 9.2.3 Odd-Page Header

.OH [arg]

The .OH macro is the same as the .EH except that it applies to odd-numbered pages.

### 9.2.4 Page Footer

.PF [arg]

The .PF macro specifies the line that is to appear at the bottom of each page. Its initial value is a blank line. If the -rCn flag is specified on the command line {2.4}, the type of copy follows the footer on a separate line. In particular, if -rC3 or -rC4 (DRAFT) is specified, the footer is initialized to contain the date {6.8} instead of being a blank line.

### 9.2.5 Even-Page Footer

.EF [arg]

The .EF macro supplies a line to be printed at the bottom of each even-numbered page immediately preceding the footer. Initial value is a blank line.

### 9.2.6 Odd-Page Footer

.OF [arg]

The .OF macro is the same as .EF except that it applies to odd-numbered pages.

### 9.2.7 First Page Footer

By default, the first page footer is a blank line. If, in the input text file, the user specifies .PF and/or .OF before the end of the first page of the document, these lines will appear at the bottom of the first page.

The header (whatever its contents) replaces the footer on the first page only if the -rN1 flag is specified on the command line {2.4}.

## 9.3 Default Header and Footer With Section-Page Numbering

Pages can be numbered sequentially within sections by "section-number page-number" {4.5}. To obtain this numbering style, -rN3 or -rN5 is specified on the command line. In this case, the default *footer* is a centered "section-page" number, e.g., 7-2; and the default page header is blank.

## 9.4 Strings and Registers in Header and Footer Macros

String and register names may be placed in arguments to header and footer macros. If the value of the string or register is to be computed when the respective header or footer is printed, invocation must be escaped by four backslashes. This is because string or register invocation will be processed three times:

1. As the argument to the header or footer macro
2. In a formatting request within the header or footer macro



3. In a .tl request during header or footer processing.

For example, page number register *P* must be escaped with four backslashes in order to specify a header in which the page number is to be printed at the right margin, e.g.:

```
.PH " "Page \\\nP"
```

creates a right-justified header containing the word "Page" followed by the page number. Similarly, to specify a footer with the "section-page" style, the user specifies (see paragraph {4.2.2.5} for meaning of *H1*):

```
.PF " " - \\\n(H1-\\n -' "
```

If the user arranges for the string *a*] to contain the current section heading which is to be printed at the bottom of each page, the .PF macro call would be:

```
.PF " "\\\*(a]"
```

If only one or two backslashes were used, the footer would print a constant value for *a*], namely, its value when .PF appeared in the input text.

### 9.5 Header and Footer Example

The following sequence specifies blank lines for header and footer lines, page numbers on the outside margin of each page (i.e., top left margin of even pages and top right margin of odd pages), and "Revision 3" on the top inside margin of each page (nothing is specified for the center):

```
.PH ""
.PF ""
.EH " \\\nP"Revision 3"
.OH " 'Revision 3' \\\nP"
```

### 9.6 Generalized Top-of-Page Processing

**Note:** This part is intended only for users accustomed to writing formatter macros.

During header processing, MM invokes two user-definable macros:

- The .TP (top of page) macro is invoked in the environment (refer to .ev request) of the header.
- The .PX is a page header user-exit macro that is invoked (without arguments) when the normal environment has been restored and with the "no-space" mode already in effect.

The effective initial definition of .TP (after the first page of a document) is

```
.de TP
.sp 3
.tl \\\*(}t
.if e 'tl \\\*(}e
.if o 'tl \\\*(}o
.sp 2
```

The string *t* contains the header, the string *e* contains the even-page header, and the string *o* contains the odd-page header as defined by the .PH, .EH, and .OH macros, respectively. To obtain more specialized page titles, the user may redefine the .TP macro to cause the desired header processing {12.5}. Formatting done within



the .TP macro is processed in an environment different from that of the body. For example, to obtain a page header that includes three centered lines of data, i.e., document number, issue date, and revision date, the user could define the .TP as follows:

```
.de TP
.sp
.ce 3
777-888-999
Iss. 2, AUG 1977
Rev. 7, SEP 1977
.sp
..
```

The .PX macro may be used to provide text that is to appear at the top of each page after the normal header and that may have tab stops to align it with columns of text in the body of the document.

### 9.7 Generalized Bottom-of-Page Processing

```
.BS
zero or more lines of text
.BE
```

Lines of text that are specified between the .BS (bottom-block start) and .BE (bottom-block end) macros will be printed at the bottom of each page after the footnotes (if any) but before the page footer. This block of text is removed by specifying an empty block, i.e.:

```
.BS
.BE
```

The bottom block will appear on the table of contents, pages, and the cover sheet for memorandum for file, but not on the technical memorandum or released-paper cover sheets.

### 9.8 Top and Bottom (Vertical) Margins

```
.VM [top] [bottom]
```

The .VM (vertical margin) macro allows the user to specify additional space at the top and bottom of the page. This space precedes the page header and follows the page footer. The .VM macro takes two unscaled arguments that are treated as v's. For example:

```
.VM 10 15
```

adds 10 blank lines to the default top of page margin and 15 blank lines to the default bottom of page margin. Both arguments must be positive (default spacing at the top of the page may be decreased by redefining .TP).

### 9.9 Proprietary Marking

```
.PM [code]
```

The .PM (proprietary marking) macro appends to the page footer a PRIVATE, NOTICE, BELL LABORATORIES PROPRIETARY, or BELL LABORATORIES RESTRICTED disclaimer. The *code* argument may be:

|             |                                      |
|-------------|--------------------------------------|
| <i>code</i> | <i>Disclaimer</i>                    |
| <i>none</i> | turn off previous disclaimer, if any |



| <i>code</i> | <i>Disclaimer</i>             |
|-------------|-------------------------------|
| P           | PRIVATE                       |
| N           | NOTICE                        |
| BP          | BELL LABORATORIES PROPRIETARY |
| BR          | BELL LABORATORIES RESTRICTED  |

These disclaimers are in a form approved for use by the Bell System. The user may alternate disclaimers by use of the .BS/.BE macro pair.

### 9.10 Private Documents

.nr *Pv* value

The word "PRIVATE" may be printed, centered, and underlined on the second line of a document (preceding the page header). This is done by setting the *Pv* register value:

| <i>value</i> | <i>Meaning</i>                 |
|--------------|--------------------------------|
| 0            | do not print PRIVATE (default) |
| 1            | PRIVATE on first page only     |
| 2            | PRIVATE on all pages           |

If *value* is 2, the user definable .TP macro may not be used because the .TP macro is used by MM to print "PRIVATE" on all pages except the first page of a memorandum on which .TP is not invoked.

### 10. Table of Contents and Cover Sheet

The table of contents and the cover sheet for a document are produced by invoking the .TC and .CS macros, respectively.

**Note:** This section refers to cover sheets for technical memoranda and released papers only. The mechanism for producing a memorandum for file cover sheet was discussed earlier {6.5}.

These macros normally appear once at the end of the document, after the Signature Block {6.11.1} and Notations {6.11.2} macros, and may occur in either order.

The table of contents is produced at the end of the document because the entire document must be processed before the table of contents can be generated. Similarly, the cover sheet may not be desired by a user and is therefore produced at the end.

#### 10.1 Table of Contents

.TC [*slevel*] [*spacing*] [*tlevel*] [*tab*] [*head1*] [*head2*] [*head3*] [*head4*] [*head5*]

The .TC macro generates a table of contents containing heading levels that were saved for the table of contents as determined by the value of the *Cl* register {4.4}. Arguments to .TC control spacing before each entry, placement of associated page number, and additional text on the first page of the table of contents before the word "CONTENTS".

Spacing before each entry is controlled by the first and second arguments ([*slevel*] and [*spacing*]). Headings whose level is less than or equal to *slevel* will have *spacing* blank lines (halves of a vertical space) before them. Both *slevel* and *spacing* default to 1. This means that first-level headings are preceded by one blank line (one-half a vertical space). The *slevel* argument does not control what levels of heading have been saved; saving of headings is the function of the *Cl* register.



The third and fourth arguments (*[tlevel]* and *[tab]*) control placement of associated page number for each heading. Page numbers can be justified at the right margin with either blanks or dots (called leaders) separating the heading text from the page number, or the page numbers can follow the heading text.

For headings whose level is less than or equal to *tlevel* (default 2), page numbers are justified at the right margin. In this case, the value of *tab* determines the character used to separate heading text from page number. If *tab* is 0 (default value), dots (i.e., leaders) are used. If *tab* is greater than 0, spaces are used.

For headings whose level is greater than *tlevel*, page numbers are separated from heading text by two spaces (i.e., page numbers are "ragged right", not right justified).

Additional arguments (*[head1]* ... *[head5]*) are horizontally centered on the page and precede the table of contents.

If the .TC macro is invoked with at most four arguments, the user-exit macro .TX is invoked (without arguments) before the word "CONTENTS" is printed, or the user-exit macro .TY is invoked and the word "CONTENTS" is not printed.

By defining .TX or .TY and invoking .TC with at most four arguments, the user can specify what needs to be done at the top of the first page of the table of contents. For example:

```
.de TX
.ce 2
Special Application
Message Transmission
.sp 2
.in +10n
Approved: \l'3i'
.in
.sp
..
.TC
```

yields the following output when the file is formatted

```
Special Application
Message Transmission
```

Approved: \_\_\_\_\_

## CONTENTS

If the .TX macro were defined as .TY, the word "CONTENTS" would be suppressed. Defining .TY as an empty macro will suppress "CONTENTS" with no replacement:

```
.de TY
..
```

By default, the first level headings will appear in the table of contents left justified. Subsequent levels will be aligned with the text of headings at the preceding level. These indentations may be changed by defining the



*Ci* string which takes a maximum of seven arguments corresponding to the heading levels. It must be given at least as many arguments as are set by the *CI* register. Arguments must be scaled; for example, with "*CI* = 5":

```
.ds Ci .25i .5i .75i 1i 1i
```

or

```
.ds Ci 0 2n 4n 6n 8n
```

Two other registers are available to modify the format of the table of contents — *Oc* and *Cp*. By default, table of contents pages will have lowercase Roman numeral page numbering. If the *Oc* register is set to 1, the .TC macro will not print any page number but will instead reset the *P* register to 1. It is the user's responsibility to give an appropriate page footer to specify the placement of the page number. Ordinarily, the same .PF macro (page footer) used in the body of the document will be adequate.

The list of figures, tables, etc. pages will be produced separately unless *Cp* is set to 1, which causes these lists to appear on the same page as the table of contents.

## 10.2 Cover Sheet

```
.CS [pages] [other] [total] [figs] [tbls] [refs]
```

The .CS macro generates a cover sheet in either the released paper or technical memorandum style (see paragraph {6.5} for details of the memorandum for file cover sheet). All other information for the cover sheet is obtained from data given before the .MT macro call {6.1}. If the technical memorandum style is used, the .CS macro generates the "Cover Sheet for Technical Memorandum". The data that appear in the lower left corner of the technical memorandum cover sheet (counts of: pages of text, other pages, total pages, figures, tables, and references) are generated automatically (0 is used for "other pages"). These values may be changed by supplying the corresponding arguments to the .CS macro. If the released-paper style is used, all arguments to .CS are ignored.

## 11. References

There are two macros (.RS and .RF) that delimit the text of references, a string that automatically numbers the subsequent references, and an optional macro (.RP) that produces reference pages within the document.

### 11.1 Automatic Numbering of References

Automatically numbered references may be obtained by typing \\*(*Rf*) (invoking the string *Rf*) immediately after the text to be referenced. This places the next sequential reference number (in a smaller point size) enclosed in brackets one-half line above the text to be referenced. Reference count is kept in the *Rf* number register.

### 11.2 Delimiting Reference Text

```
.RS [string-name]
.RF
```

The .RS and .RF macros are used to delimit text of each reference as shown below:

```
A line of text to be referenced.\*(Rf)
.RS
reference text
.RF
```



### 11.3 Subsequent References

The **.RS** macro takes one argument, a *string-name*. For example:

```
.RS aA
reference text
.RF
```

The string *aA* is assigned the current reference number. This string may be used later in the document as the string call, **\\*(aA**, to reference text which must be labeled with a prior reference number. The reference is output enclosed in brackets one-half line above the text to be referenced. No **.RS/.RF** pair is needed for subsequent references.

### 11.4 Reference Page

```
.RP [arg1] [arg2]
```

A reference page, entitled by default "References", will be generated automatically at the end of the document (before table of contents and cover sheet) and will be listed in the table of contents. This page contains the reference items (i.e., reference text enclosed within **.RS/.RF** pairs). Reference items will be separated by a space (one-half a vertical space) unless the *Ls* register is set to 0 to suppress this spacing. The user may change the reference page title by defining the *Rp* string:

```
.ds Rp " New Title "
```

The **.RP** (reference page) macro may be used to produce reference pages anywhere else within a document (i.e., after each major section). It is not needed to produce a separate reference page with default spacings at the end of the document.

Two **.RP** macro arguments allow the user to control resetting of reference numbering and page skipping.

| <i>arg1</i> | <i>Meaning</i> |
|-------------|----------------|
|-------------|----------------|

|   |                                   |
|---|-----------------------------------|
| 0 | reset reference counter (default) |
| 1 | do not reset reference counter    |

| <i>arg2</i> | <i>Meaning</i> |
|-------------|----------------|
|-------------|----------------|

|   |                                     |
|---|-------------------------------------|
| 0 | put on separate page (default)      |
| 1 | do not cause a following <b>.SK</b> |
| 2 | do not cause a preceding <b>.SK</b> |
| 3 | no <b>.SK</b> before or after       |

If no **.SK** is issued by the **.RP** macro, a single blank line will separate the references from the following/preceding text. The user may wish to adjust spacing. For example, to produce references at the end of each major section:

```
.sp 3
.RP 1 2
.H 1 " Next Section "
```

## 12. Miscellaneous Features

### 12.1 Bold, Italic, and Roman Fonts

```
.B [bold-arg] [previous-font-arg] ...
```



```
.I [italic-arg] [previous-font-arg] ...
.R
```

When called without arguments, the .B macro changes the font to bold and the .I macro changes to underlining (italic). This condition continues until the occurrence of the .R macro which causes the Roman font to be restored. Thus:

```
.I
here is some text.
.R
```

yields underlined text via the **nroff** and italic text via the **troff(1)** formatter.

If the .B or .I macro is called with one argument, that argument is printed in the appropriate font (underlined in the **nroff** formatter for .I). Then the previous font is restored (underlining is turned off in the **nroff** formatter). If two or more arguments (maximum six) are given with a .B or .I macro call, the second argument is concatenated to the first with no intervening space (1/12 space if the first font is italic) but is printed in the previous font. Remaining pairs of arguments are similarly alternated. For example:

```
.I italic "text" right-justified
```

produces

```
italic text right-justified
```

The .B and .I macros alternate with the prevailing font at the time the macros are invoked. To alternate specific pairs of fonts, the following macros are available:

```
.IB .BI .IR .RI .RB .BR
```

Each macro takes a maximum of six arguments and alternates arguments between specified fonts.

When using a terminal that cannot underline, the following can be inserted at the beginning of the document to eliminate all underlining:

```
.rm ul
.rm cu
```

**Note:** Font changes in headings are handled separately {4.2.2.4.1}.

## 12.2 Justification of Right Margin

```
.SA [arg]
```

The .SA macro is used to set right-margin justification for the main body of text. Two justification flags are used—*current* and *default*. The ".SA 0" call sets both flags to no justification; it acts like the .na request. The ".SA 1" call sets both flags to cause both right and left justification, the same as the .ad request. However, calling .SA without an argument causes the current flag to be copied from the default flag, thus performing either a .na or .ad depending on the *default*. Initially, both flags are set for no justification in the **nroff** formatter and for justification in the **troff** formatter.

In general, the no adjust request (.na) can be used to ensure that justification is turned off, but .SA should be used to restore justification, rather than the .ad request. In this way, justification or no justification for the remainder of the text is specified by inserting ".SA 0" or ".SA 1" once at the beginning of the document.



### 12.3 SCCS Release Identification

The *RE* string contains the SCCS release and the MM text formatting macro package current version level. For example:

This is version \\*(RE of the macros.

produces

This is version 10.129 of the macros.

This information is useful in analyzing suspected bugs in MM. The easiest way to have the release identification number appear in the output is to specify `-rD1 {2.4}` on the command line. This causes the *RE* string to be output as part of the page header {9.2.1}.

### 12.4 Two-Column Output

.2C  
text and formatting requests (except another .2C)  
.1C

The MM text formatting macro package can format two columns on a page. The .2C macro begins 2-column processing which continues until a .1C macro (1-column processing) is encountered. In 2-column processing, each physical page is thought of as containing 2-columnar "pages" of equal (but smaller) "page" width. Page headers and footers are not affected by 2-column processing. The .2C macro does not balance 2-column output.

It is possible to have full-page width footnotes and displays when in 2-column mode, although default action is for footnotes and displays to be narrow in 2-column mode and wide in 1-column mode. Footnote and display width is controlled by the .WC (width control) macro, which takes the following arguments:

| arg | Meaning                                                                                                                         |
|-----|---------------------------------------------------------------------------------------------------------------------------------|
| N   | Default mode ( <code>-WF</code> , <code>-FF</code> , <code>-WD</code> , <code>FB</code> ).                                      |
| WF  | Wide footnotes (even in 2-column mode).                                                                                         |
| -WF | DEFAULT: Turn off WF. Footnotes follow column mode; wide in 1-column mode (1C), narrow in 2-column mode (2C), unless FF is set. |
| FF  | First footnote. All footnotes have same width as first footnote encountered for that page.                                      |
| -FF | DEFAULT: Turn off FF. Footnote style follows settings of WF or -WF.                                                             |
| WD  | Wide displays (even in 2-column mode).                                                                                          |
| -WD | DEFAULT: Displays follow the column mode in effect when display is encountered.                                                 |
| FB  | DEFAULT: Floating displays cause a break when output on the current page.                                                       |
| -FB | Floating displays on current page do not cause a break.                                                                         |

**Note:** The ".WC WD FF" command will cause all displays to be wide and all footnotes on a page to be the same width while ".WC N" will reinstate default actions. If conflicting settings are given to .WC, the last one is used. A ".WC WF -WF" command has the effect of a ".WC -WF".



### 12.5 Column Headings for Two-Column Output

**Note:** This section is intended only for users accustomed to writing formatter macros.

In 2-column processing output, it is sometimes necessary to have headers over each column, as well as headers over the entire page. This is accomplished by redefining the .TP macro {9.6} to provide header lines both for the entire page and for each of the columns. For example:

```
.de TP
.sp 2
.tl 'Page \\nP*OVERALL"
.tl "TITLE"
.sp
.nf
.ta 16C 31R 34 50C 65R
leftØcenterØrightØleftØcenterØright
Øfirst columnØØsecond column
.fi
.sp 2
..
```

where Ø stands for the tab character.

The above example will produce two lines of page header text plus two lines of headers over each column. Tab stops are for a 65-en overall line length.

### 12.6 Vertical Spacing

.SP [lines]

There exists several ways of obtaining vertical spacing, all with different effects. The .sp request spaces the number of lines specified unless the no space (.ns) mode is on, then the .sp request is ignored. The no space mode is set at the end of a page header to eliminate spacing by a .sp or .bp request that happens to occur at the top of a page. This mode can be turned off by the .rs (restore spacing) request.

The .SP macro is used to avoid the accumulation of vertical space by successive macro calls. Several .SP calls in a row will not produce the sum of the arguments but only the maximum argument. For example, the following produces only three blank lines:

```
.SP 2
.SP 3
.SP
```

Many MM macros utilize .SP for spacing. For example, ".LE 1" {5.1.3} immediately followed by ".P" {4.1} produces only a single blank line (one-half a vertical space) between the end of the list and the following paragraph. An omitted argument defaults to one blank line (one vertical space). Negative arguments are not permitted. The argument must be unscaled but fractional amounts are permitted. The .SP macro (as well as .sp) is also inhibited by the .ns request.

### 12.7 Skipping Pages

.SK [pages]

The .SK macro skips pages but retains the usual header and footer processing. If the pages argument is omitted, null, or 0, .SK skips to the top of the next page unless it is currently at the top of a page (then it does



nothing). The ".SK *n*" skips *n* pages. The .SK macro always positions text that follows it at the top of a page, while ".SK 1" always leaves one page blank except for the header and footer.

### 12.8 Forcing an Odd Page

.OP

The .OP macro is used to ensure that formatted output text following the macro begins at the top of an odd-numbered page. If currently at the top of an odd-numbered page, text output begins on that page (no motion takes place). If currently on an even page, text resumes printing at the top of the next page. If currently on an odd page (but not at the top of the page), one blank page is produced, and printing resumes on the next odd-numbered page after that.

### 12.9 Setting Point Size and Vertical Spacing

.S [point size] [vertical spacing]

In the troff formatter, the default point size (obtained from the MM register *S*{2.4}) is 10 points, and the vertical spacing is 12 points (six lines per inch). Prevailing point size and vertical spacing may be changed by invoking the .S macro.

The mnemonics D (default value), C (current value), and P (previous value) may be used for both arguments.

- If an argument is *negative*, current value is decremented by the specified amount.
- If an argument is *positive*, current value is incremented by the specified amount.
- If an argument is *unsigned*, it is used as the new value.
- If there are no arguments, the .S macro defaults to P.
- If the first argument is specified but the second is not, then (default) D is used for the vertical spacing.

Default value for vertical spacing is always two points greater than the current point size. Footnotes {8} are two points smaller than the body with an additional 3-point space between footnotes. A null ( " " ) value for either argument defaults to C (current value). Thus, if *n* is a numeric value:

```
.S          =.S P P
.S " " n    =.S C n
.S n " "    =.S n C
.S n        =.S n D
.S " "      =.S C D
.S " " " "  =.S C C
.S n n      =.S n n
```

If the first argument is greater than 99, the default point size (10 points) is restored. If the second argument is greater than 99, the default vertical spacing (current point size plus two points) is used. For example:

```
.S 100      =.S 10 12
.S 14 111   =.S 14 16
```

The .SM macro allows the user to reduce by one point the size of a string:

.SM string1 [string2] [string3]



If the third argument is omitted, the first argument is made smaller and is concatenated with the second argument if the latter is specified. If all three arguments are present (even if any are null), the second argument is made smaller and all three arguments are concatenated. For example:

| <i>INPUT</i>  | <i>OUTPUT</i> |
|---------------|---------------|
| .SM X         | X             |
| .SM X Y       | XY            |
| .SM Y X Y     | YXY           |
| .SM YXYX      | YXYX          |
| .SM YXYX )    | YXYX)         |
| .SM ( YXYX )  | (YXYX)        |
| .SM Y XYX " " | YXYX          |

### 12.10 Producing Accents

The following strings may be used to produce accents for letters:

|                   | <i>INPUT</i> | <i>OUTPUT</i> |
|-------------------|--------------|---------------|
| Grave accent      | c\`          | ç             |
| Acute accent      | e\`          | é             |
| Circumflex        | o\`          | ô             |
| Tilde             | n\`          | ñ             |
| Cedilla           | c\`          | ç             |
| Lower-case umlaut | u\`          | ü             |
| Upper-case umlaut | U\`          | Ü             |

### 12.11 Inserting Text Interactively

.RD [prompt] [diversion] [string]

The .RD (read insertion) macro allows a user to stop the standard output of a document and to read text from the standard input until two consecutive newline characters are found. When newline characters are encountered, normal output is resumed.

- The *prompt* argument will be printed at the terminal. If not given, .RD signals the user with a BEL on terminal output.
- The *diversion* argument allows the user to save all text typed in after the prompt in a macro whose name is that of the diversion.
- The *string* argument allows the user to save for later reference the first line following the prompt in the named string.

The .RD macro follows the formatting conventions in effect. Thus, the following examples assume that the .RD is invoked in no-fill mode (.nf):

.RD Name aA bB



produces

Name: J. Jones (user types name)  
16 Elm Rd.,  
Piscataway

The diverted macro .aA will contain

J. Jones  
16 Elm Rd.,  
Piscataway

The string `bB(\*bB)` contains "J. Jones".

A newline character followed by an EOF (user specifiable CONTROL d) also allows the user to resume normal output.

### 13. Errors and Debugging

#### 13.1 Error Terminations

When a macro detects an error, the following actions occur:

- A break occurs.
- The formatter output buffer (which may contain some text) is printed to avoid confusion regarding location of the error.
- A short message is printed giving the name of the macro that detected the error, type of error, and approximate line number in the current input file of the last processed input line. Error messages are explained in Table 4.D.
- Processing terminates unless register D {2.4} has a positive value. In the latter case, processing continues even though the output is guaranteed to be deranged from that point on.

The error message is printed by outputting the message directly to the user terminal. If an output filter, such as 300(1), 450(1), or `hp(1)` is being used to post-process the `nroff` formatter output, the message may be garbled by being intermixed with text held in that filter's output buffer.

**Note:** If any of `cw(1)`, `eqn(1)/neqn`, and `tbl(1)` programs are being used and if the `-olist` option of the formatter causes the last page of the document not to be printed, a harmless "broken pipe" message may result.

#### 13.2 Disappearance of Output

Disappearance of output usually occurs because of an unclosed diversion (e.g., a missing `.DE` or `.FE` macro). Fortunately, macros that use diversions are careful about it, and these macros check to make sure that illegal nestings do not occur. If any error message is issued concerning a missing `.DE` or `.FE`, the appropriate action is to search backwards from the termination point looking for the corresponding associated `.DF`, `.DS`, or `.FS` (since these macros are used in pairs).

The following command:

```
grep -n '^\[EDFRT\][EFNQS]' files ...
```



prints all the .DF, .DS, .DE, .EQ, .EN, .FS, .FE, .RS, .RF, .TS, and .TE macros found in *files ...*, each preceded by its file name and the line number in that file. This listing can be used to check for illegal nesting and/or omission of these macros.

## 14. Extending and Modifying MM Macros

### 14.1 Naming Conventions

In this part, the following conventions are used to describe names:

- n: Digit
- a: Lowercase letter
- A: Uppercase letter
- x: Any alphanumeric character (: a, n, A, or 1, ie., letter or digit)
- s: any nonalphanumeric character (special character)

All other characters are literals (i.e., characters stand for themselves).

Request, macro, and string names are kept by the formatters in a single internal table; therefore, there must be no duplication among such names. Number register names are kept in a separate table.

#### 14.1.1 Names Used by Formatters

- requests: aa (most common)
- an (only one, currently: c2)
- registers: aa (normal)
- .x (normal)
- .s (only one, currently: .s)
- a. (only one, currently: c.)
- % (page number)

#### 14.1.2 Names Used by MM

- macros and strings: A, AA, Aa (accessible to users; e.g., macros P and HU, strings F, BU, and Lt)
- nA (accessible to users; only two, currently: 1C and 2C)
- aA (accessible to users; only one, currently: nP)
- s (accessible to users; only the seven accents, currently {12.10})
- )x, }x, ]x, >x, ?x (internal)
- registers: An, Aa (accessible to users; e.g., H1, Fg)
- A (accessible to users; meant to be set on the command line; e.g., C)
- :x, ;x, #x, ?x, !x (internal)



### 14.1.3 Names Used by CW, EQN/NEQN, and TBL Programs

The `cw(1)` program is the constant-width font preprocessor for the `troff` formatter. It uses the following five macro names:

`.CD`, `.CN`, `.CP`, `.CW`, and `.PC`.

This preprocessor also uses the number register names `cE` and `cW`. Mathematical equation preprocessors, `eqn(1)` and `neqn`, use registers and string names of the form `nn`. The table preprocessor, `tbl(1)`, uses `T&`, `T#`, and `TW`, and names of the form:

`a-` `a+` `a!` `nn` `na` `^a` `#a` `#s`

### 14.1.4 Names Defined by User

Names that consist either of a single lowercase letter or a lowercase letter followed by a character other than a lowercase letter (names `.c2` and `.nP` are already used) should be used to avoid duplication with already used names. The following is a possible naming convention:

|            |                                                                           |
|------------|---------------------------------------------------------------------------|
| macros:    | <code>aA</code> (e.g., <code>bG</code> , <code>kW</code> )                |
| strings:   | <code>as</code> (e.g., <code>c</code> , <code>f</code> , <code>p</code> ) |
| registers: | <code>a</code> (e.g., <code>f</code> , <code>t</code> )                   |

## 14.2 Sample Extensions

### 14.2.1 Appendix Headings

The following is a way of generating and numbering appendix headings:

```
.nr Hu 1
.nr a 0
.de aH
.nr a +1
.nr P 0
.PH " "Appendix \\na-\\\\\\\\\\\\\\nP' "
.SK
.HU " \\$1 "
```

After the above initialization and definition, each call of the form `.aH " title"` begins a new page (with the page header changed to "Appendix *a-n*") and generates an unnumbered heading of *title*, which, if desired, can be saved for the table of contents. Those who wish appendix titles to be centered must, in addition, set the register `Hc` to 1 {4.2.2.3}.

### 14.2.2 Hanging Indent With Tabs

The following example illustrates the use of the hanging indent feature of variable-item lists {5.1.1.6}. A user-defined macro is defined to accept four arguments that make up the *mark*. In the output, each argument is to be separated from the previous one by a tab; tab settings are defined later. Since the first argument may begin with a period or apostrophe, the `"\&"` is used so that the formatter will not interpret such a line as a formatter request or macro call.

**Note:** The 2-character sequence `"\&"` is understood by formatters to be a "zero-width" space. It causes no output characters to appear, but it removes the special meaning of a leading period or apostrophe.



The “\t” is translated by the formatter into a tab. The “\c” is used to concatenate the input text that follows the macro call to the line built by the macro. The macro and an example of its use are:

```
.de aX
.LI
\&\\$1\t\\$2\t\\$3\t\\$4\t\c
```

```
ta 8 14 20 24
```

```
VL 36
```

```
.aX .nh off \- no
```

No hyphenation.

Automatic hyphenation is turned off.

Words containing hyphens

(e.g., mother-in-law) may still be split across lines.

```
.aX .hy on \- no
```

Hyphenate.

Automatic hyphenation is turned on.

```
.aX .hc\<sp>c none none no
```

Hyphenation indicator character is set to “c” or removed.

During text processing, the indicator is suppressed and will not appear in the output.

Prepending the indicator to a word has the effect of preventing hyphenation of that word.

```
.LE
```

where <sp> stands for a space.

The resulting output is:

|       |      |      |    |                                                                                                                                                                                                                                            |
|-------|------|------|----|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| .nh   | off  | —    | no | No hyphenation. Automatic hyphenation is turned off. Words containing hyphens (e.g., mother-in-law) may still be split across lines.                                                                                                       |
| .hy   | on   | —    | no | Hyphenate. Automatic hyphenation is turned on.                                                                                                                                                                                             |
| .hc c | none | none | no | Hyphenation indicator character is set to “c” or removed. During text processing, the indicator is suppressed and will not appear in the output. Prepending the indicator to a word has the effect of preventing hyphenation of that word. |



## 15. Summary

The following are qualities of MM that have been emphasized in its design in approximate order of importance:

- *Robustness in the face of error*—A user need not be an **nroff**/**troff** expert to use MM macros. When the input is incorrect, either the macros attempt to make a reasonable interpretation of the error or an error message describing the error is produced. An effort has been made to minimize the possibility that a user would get cryptic system messages or strange output as a result of simple errors.
- *Ease of use for simple documents*—It is not necessary to write complex sequences of commands to produce documents. Reasonable macro argument default values are provided where possible.
- *Parameterization*—There are many different preferences in the area of document styling. Many parameters are provided so that users can adapt input text files to produce output documents to their respective needs over a wide range of styles.
- *Extension by moderately expert users*—A strong effort has been made to use mnemonic naming conventions and consistent techniques in construction of macros. Naming conventions are given so that a user can add new macros or redefine existing ones if necessary.
- *Device independence*—A common use of MM is to produce documents on hard copy via teletypewriter terminals using the **nroff** formatter. Macros can be used conveniently with both 10- and 12-pitch terminals. In addition, output can be displayed on an appropriate CRT terminal. Macros have been constructed to allow compatibility with the **troff**(1) formatter so that output can be produced on both a phototypesetter and a teletypewriter/CRT terminal.
- *Minimization of input*—The design of macros attempts to minimize repetitive typing. For example, if a user wants to have a blank line after all first- or second-level headings, the user need only set a specific parameter once at the beginning of a document rather than type a blank line after each such heading.
- *Decoupling of input format from output style*—There is but one way to prepare the input text although the user may obtain a number of output styles by setting a few global flags. For example, the **.H** macro is used for all numbered headings, yet the actual output style of these headings may be made to vary from document to document or within a single document.



**INPUT:**

.ND "May 31, 1979"  
.TL 334455  
Out-of-Hours Course Description  
.AU "D. W. Stevenson" DWS PY 9876 5432 1X-123  
.MT 0  
.DS  
J. M. Jones:  
.DE  
.P  
Please use the following description for the out-of-hours course  
.I  
Document Preparation on the UNIX\*  
.R  
.FS \*  
Trademark of Bell Laboratories.  
.FE  
.I "Time-Sharing Operating System:"  
.P  
The course is intended for clerks, typists, and others  
who intend to use the UNIX system for preparing documentation.  
The course will cover such topics as:  
.VL 18  
.LI Environment:  
utilizing a time-sharing computer system;  
accessing the system; using appropriate output terminals.  
.LI Files:  
how text is stored on the system;  
directories; manipulating files.  
.LI "Text editing:"  
how to enter text so that subsequent revisions are easier to make;  
how to use the editing system to add, delete, and move lines of text;  
how to make corrections.  
.LI "Text processing:"  
basic concepts;  
use of general purpose formatting packages.  
.LI "Other facilities:"  
additional capabilities useful to the typist such as the  
.I "spell, diff,"  
and  
.I grep  
commands, and a desk-calculator package.  
.LE  
.SG jrm  
.NS 0  
S. P. Ienane  
H. O. Del  
M. Hill  
.NE

Fig. 4.1 — Examples of a Simple Letter (Sheet 1 of 3)



**nrff OUTPUT:****Bell Laboratories**

subject: Out-of-Hours Course Description -  
Case 334455

date: May 31, 1979

from: D. W. Stevenson  
PY 9876  
1X-123 x5432

J. M. Jones:

Please use the following description for the out-of-hours course  
Document Preparation on the UNIX\* Time-Sharing Operating System:

The course is intended for clerks, typists, and others who intend to use the UNIX system for preparing documentation. The course will cover such topics as:

Environment: utilizing a time-sharing computer system; accessing the system; using appropriate output terminals.

Files: how text is stored on the system; directories; manipulating files.

Text editing: how to enter text so that subsequent revisions are easier to make; how to use the editing system to add, delete, and move lines of text; how to make corrections.

Text processing: basic concepts; use of general-purpose formatting packages.

Other facilities: additional capabilities useful to the typist such as the spell, diff, and grep commands, and a desk-calculator package.

PY-9876-DWS-jrm

D. W. Stevenson

Copy to  
S. P. Lename  
H. O. Del  
M. Hill

---

\* Trademark of Bell Laboratories.

Fig. 4.1 — Examples of a Simple Letter (Sheet 2 of 3)



**Bell Laboratories**subject: **Out-of-Hours Course Description - Case 334455**date: **May 31, 1979**from: **D. W. Stevenson**  
**PY 9876**  
**1X-123 x5432****J. M. Jones:**

Please use the following description for the Out-of-Hours course *Document Preparation on the UNIX\* Time-Sharing System*:

The course is intended for clerks, typists, and others who intend to use the UNIX system for preparing documentation. The course will cover such topics as:

- Environment:** utilizing a time-sharing computer system; accessing the system; using appropriate output terminals.
- Files:** how text is stored on the system; directories; manipulating files.
- Text editing:** how to enter text so that subsequent revisions are easier to make; how to use the editing system to add, delete, and move lines of text; how to make corrections.
- Text processing:** basic concepts; use of general-purpose formatting packages.
- Other facilities:** additional capabilities useful to the typist such as the *spell*, *diff*, and *grep* commands, and a desk-calculator package.

**PY-9876-DWS-jrm****D. W. Stevenson**

Copy to  
S. P. Lename  
H. O. Del  
M. Hill

---

\* Trademark of Bell Laboratories.



- 1 -

**INPUT:**

.P  
.FD 10  
This example illustrates several footnote styles for both labeled and automatically numbered footnotes. With the footnote style set to the NROFF default, process the first footnote.\\*F  
.FS  
This is the first footnote text example (.FD 10). This is the default style for NROFF. The right margin is not justified. Hyphenation is not permitted. Text is indented, and the automatically generated label is right justified in the text-indent space.  
.FE  
and follow it by a second footnote.\*\*\*\*\*  
.FS \*\*\*\*\*  
This is the second footnote text example (.FD 10). This is also the default NROFF style but with a long footnote label (\*\*\*\*\*) provided by the user.  
.FE  
.FD 1  
Footnote style is changed by using the .FD macro to specify hyphenation, right margin justification, indentation, and left justification of the label. This produces the third footnote.\\*F  
.FS  
This is the third footnote example (.FD 1). The right margin is justified, the footnote text is indented, and the label is left justified in the text-indent space. Although not necessarily illustrated by this example, hyphenation is permitted.  
.FE  
and then the fourth footnote.\(dg  
.FS †  
This is the fourth footnote example (.FD 1). The style is the same as the third footnote.  
.FE  
.FD 6  
Footnote style is set again via the .FD macro for no hyphenation, no right margin justification, no indentation, and with the label left justified. This produces the fifth footnote.\\*F  
.FS  
This is the fifth footnote example (.FD 6). The right margin is not justified, hyphenation is not permitted, footnote text is not indented, and the label is placed at the beginning of the first line.  
.FE

Fig. 4.2 — Examples of Footnotes (Sheet 1 of 2)



- 2 -

**OUTPUT:**

This example illustrates several footnote styles for both labeled and automatically numbered footnotes. With the footnote style set to the NROFF default, process the first footnote<sup>1</sup> and follow it by a second footnote.\*\*\*\*\* Footnote style is changed by using the .FD macro to specify hyphenation, right margin justification, indentation, and left justification of the label. This produces the third footnote,<sup>2</sup> and then the fourth footnote.<sup>+</sup> Footnote style is set again via the .FD macro for no hyphenation, no right margin justification, no indentation, and with the label left justified. This produces the fifth footnote.<sup>3</sup>

---

1. This is the first footnote text example (.FD 10). This is the default style for NROFF. The right margin is not justified. Hyphenation is not permitted. Text is indented, and the automatically generated label is right justified in the text-indent space.

\*\*\*\*\* This is the second footnote text example (.FD 10). This is also the default NROFF style but with a long footnote label (\*\*\*\*\* ) provided by the user.

2. This is the third footnote example (.FD 1). The right margin is justified, the footnote text is indented, and the label is left justified in the text-indent space. Although not necessarily illustrated by this example, hyphenation is permitted.

+ This is the fourth footnote example (.FD 1). The style is the same as the third footnote.

3. This is the fifth footnote example (.FD 6). The right margin is not justified, hyphenation is not permitted, footnote text is not indented, and the label is placed at the beginning of the first line.

Fig. 4.2 — Examples of Footnotes (Sheet 2 of 2)



TABLE 4.A  
MM MACRO NAMES SUMMARY

| MACRO | DESCRIPTION {PARAGRAPH}                                                                     |
|-------|---------------------------------------------------------------------------------------------|
| 1C    | 1-column processing {12.4}<br>.1C                                                           |
| 2C    | 2-column processing {12.4}<br>.2C                                                           |
| AE    | Abstract end {6.5}<br>.AE                                                                   |
| AF    | Alternate format of "Subject/Date/From" block {6.9}<br>.AF [company-name]                   |
| AL    | Automatically incremented list start {5.1.1.1}<br>.AL [type] [text-indent] [1]              |
| AS    | Abstract start {6.5}<br>.AS [arg] [indent]                                                  |
| AT    | Author's title {6.3}<br>.AT [title] ...                                                     |
| AU    | Author information {6.3}<br>.AU name [initials] [loc] [dept] [ext] [room] [arg] [arg] [arg] |
| AV    | Approval signature {6.11.3}<br>.AV [name]                                                   |
| B     | Bold {12.1}<br>.B [bold-arg] [previous-font-arg] [bold] [prev] [bold] [prev]                |
| BE    | Bottom block end {9.7}<br>.BE                                                               |
| BI    | Bold/Italic {12.1}<br>.BI [bold-arg] [italic-arg] [bold] [italic] [bold] [italic]           |
| BL    | Bullet list start {5.1.1.2}<br>.BL [text-indent] [1]                                        |
| BR    | Bold/Roman {12.1}<br>.BR [bold-arg] [Roman-arg] [bold] [Roman] [bold] [Roman]               |
| BS    | Bottom block start {9.7}<br>.BS                                                             |
| CS    | Cover sheet {10.2}<br>.CS [pages] [other] [total] [figs] [tbls] [refs]                      |
| DE    | Display end {7.1}<br>.DE                                                                    |
| DF    | Display floating start {7.2}<br>.DF [format] [fill] [right-indent]                          |
| DL    | Dash list start {5.1.1.3}<br>.DL [text-indent] [1]                                          |



TABLE 4.A (Contd)

## MM MACRO NAMES SUMMARY

| MACRO | DESCRIPTION PARAGRAPH                                                                           |
|-------|-------------------------------------------------------------------------------------------------|
| DS    | Display static start {7.1}<br>.DS [format] [fill] [right-indent]                                |
| EC    | Equation caption {7.5}<br>.EC [title] [override] [flag]                                         |
| EF    | Even-page footer {9.2.5}<br>.EF [arg]                                                           |
| EH    | Even-page header {9.2.2}<br>.EH [arg]                                                           |
| EN    | End equation display {7.4}<br>.EN                                                               |
| EQ    | Equation display start {7.4}<br>.EQ [label]                                                     |
| EX    | Exhibit caption {7.5}<br>.EX [title] [override] [flag]                                          |
| FC    | Formal closing {6.11}<br>.FC [closing]                                                          |
| FD    | Footnote default format {8.3}<br>.FD [arg] [1]                                                  |
| FE    | Footnote end {8.2}<br>.FE                                                                       |
| FG    | Figure title {7.5}<br>.FG [title] [override] [flag]                                             |
| FS    | Footnote start {8.2}<br>.FS [label]                                                             |
| H     | Heading—numbered {4.2}<br>.H level [heading-text] [heading-suffix]                              |
| HC    | Hyphenation character {3.4}<br>.HC [hyphenation-indicator]                                      |
| HM    | Heading mark style {4.2.2.5}<br>(Arabic or Roman numerals, or letters)<br>.HM [arg1] ... [arg7] |
| HU    | Heading—unnumbered {4.3}<br>.HU heading-text                                                    |
| HX*   | Heading user exit X (before printing heading) {4.6}<br>.HX dlevel rlevel heading-text           |

\*See note at end of table.



TABLE 4.A (Contd)  
MM MACRO NAMES SUMMARY

| MACRO | DESCRIPTION PARAGRAPH                                                                                                          |
|-------|--------------------------------------------------------------------------------------------------------------------------------|
| HY*   | Heading user exit Y (before printing heading) {4.6}<br>.HY dlevel rlevel heading-text                                          |
| HZ*   | Heading user exit Z (after printing heading) {4.6}<br>.HZ dlevel rlevel heading-text                                           |
| I     | Italic (underline in the <b>nroff</b> formatter) {12.1}<br>.I [italic-arg] [previous-font-arg] [italic] [prev] [italic] [prev] |
| IB    | Italic/Bold {12.1}<br>.IB [italic-arg] [bold-arg] [italic] [bold] [italic] [bold]                                              |
| IR    | Italic/Roman {12.1}<br>.IR [italic-arg] [Roman-arg] [italic] [Roman] [italic] [Roman]                                          |
| LB    | List begin {5.2}<br>.LB text-indent mark-indent pad type [mark] [LI-space] [LB-space]                                          |
| LC    | List-status clear {5.3}<br>.LC [list-level]                                                                                    |
| LE    | List end {5.1.3}<br>.LE [1]                                                                                                    |
| LI    | List item {5.1.2}<br>.LI [mark] [1]                                                                                            |
| ML    | Marked list start {5.1.1.4}<br>.ML mark [text-indent] [1]                                                                      |
| MT    | Memorandum type {6.7}<br>.MT [type] [addressee] <i>or</i> .MT [4] [1]                                                          |
| ND    | New date {6.8}<br>.ND new-date                                                                                                 |
| NE    | Notation end {6.11.2}<br>.NE                                                                                                   |
| NS    | Notation start {6.11.2}<br>.NS [arg]                                                                                           |
| nP    | Double-line indented paragraphs {4.1}<br>.nP                                                                                   |
| OF    | Odd-page footer {9.2.6}<br>.OF [arg]                                                                                           |
| OH    | Odd-page header {9.2.3}<br>.OH [arg]                                                                                           |
| OK    | Other keywords for the Technical Memorandum cover sheet {6.6}<br>.OK [keyword] ...                                             |

\*See note at end of table.



TABLE 4.A (Contd)

## MM MACRO NAMES SUMMARY

| MACRO | DESCRIPTION                                                                           | PARAGRAPH |
|-------|---------------------------------------------------------------------------------------|-----------|
| OP    | Odd page {12.8}<br>.OP                                                                |           |
| P     | Paragraph {4.1}<br>.P [type]                                                          |           |
| PF    | Page footer {9.2.4}<br>.PF [arg]                                                      |           |
| PH    | Page header {9.2.1}<br>.PH [arg]                                                      |           |
| PM    | Proprietary marking {9.9}<br>.PM [code]                                               |           |
| PX*   | Page-header user exit {9.6}<br>.PX                                                    |           |
| R     | Return to regular (Roman) font {12.1}<br>.R                                           |           |
| RB    | Roman/bold {12.1}<br>.RB [Roman-arg] [bold-arg] [Roman] [bold] [Roman] [bold]         |           |
| RD    | Read insertion from terminal {12.11}<br>.RD [prompt] [diversion] [string]             |           |
| RF    | Reference end {11.2}<br>.RF                                                           |           |
| RI    | Roman/Italic {12.1}<br>.RI [Roman-arg] [italic-arg] [Roman] [italic] [Roman] [italic] |           |
| RL    | Reference list start {5.1.1.5}<br>.RL [text-indent] [1]                               |           |
| RP    | Produce reference page {11.4}<br>.RP [arg] [arg]                                      |           |
| RS    | Reference start {11.2}<br>.RS [string-name]                                           |           |
| S     | Set troff formatter point size and vertical spacing {12.9}<br>.S [size] [spacing]     |           |
| SA    | Set adjustment (right-margin justification) default {12.2}<br>.SA [arg]               |           |
| SG    | Signature line {6.11.1}<br>.SG [arg] [1]                                              |           |
| SK    | Skip pages {12.7}<br>.SK [pages]                                                      |           |

\*See note at end of table.



TABLE 4.A (Contd)

## MM MACRO NAMES SUMMARY

| MACRO | DESCRIPTION                                                                                                  | PARAGRAPH |
|-------|--------------------------------------------------------------------------------------------------------------|-----------|
| SM    | Make a string smaller {12.9}<br>.SM string1 [string2] [string3]                                              |           |
| SP    | Space vertically {12.6}<br>.SP [lines]                                                                       |           |
| TB    | Table title {7.5}<br>.TB [title] [override] [flag]                                                           |           |
| TC    | Table of contents {10.1}<br>.TC [slevel] [spacing] [tlevel] [tab] [head1] [head2] [head3]<br>[head4] [head5] |           |
| TE    | Table end {7.3}<br>.TE                                                                                       |           |
| TH    | Table header {7.3}<br>.TH [N]                                                                                |           |
| TL    | Title of memorandum {6.2}<br>.TL [charging-case] [filing-case]                                               |           |
| TM    | Technical Memorandum number(s) {6.4}<br>.TM [number] ...                                                     |           |
| TP*   | Top-of-page macro {9.6}<br>.TP                                                                               |           |
| TS    | Table start {7.3}<br>.TS [H]                                                                                 |           |
| TX*   | Table of contents user exit {10.1}<br>.TX                                                                    |           |
| TY*   | Table of contents user exit {10.1}<br>(suppresses "CONTENTS")<br>.TY                                         |           |
| VL    | Variable-item list start {5.1.1.6}<br>.VL text-indent [mark-indent] [1]                                      |           |
| VM    | Vertical margins {9.8}<br>.VM [top] [bottom]                                                                 |           |
| WC    | Footnote and Display Width control {12.4}<br>.WC [format]                                                    |           |

\*Macros marked with an asterisk are not, in general, called (invoked) directly by the user. They are "user exits" defined by the user and called by the MM macros from inside header, footer, or other macros.



TABLE 4.B  
STRING NAMES SUMMARY

| STRING NAME | DESCRIPTION { PARAGRAPH }                                                                                                                                                                                |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BU          | Bullet {3.7}<br>NROFF: •<br>TROFF: •                                                                                                                                                                     |
| Ci          | Table of contents indent list {10.1}<br>Up to seven <i>args</i> (must be scaled) for heading levels                                                                                                      |
| DT          | Date {6.8}<br>Current date, unless overridden<br>Month, day, year (e.g., July 16, 1982)                                                                                                                  |
| EM          | Em dash string {3.8}<br>Produces an em dash in the <i>troff</i> formatter and a double hyphen in <i>nroff</i>                                                                                            |
| F           | Footnote numberer {8.1}<br>NROFF:\u\n+(:p\d<br>TROFF:\v'-4m's-3\n+(:p\s0\v'.4m'                                                                                                                          |
| HF          | Heading font list {4.2.2.4.1}<br>Up to seven codes for heading levels 1 through 7<br>3 3 2 2 2 2 (levels 1 and 2 bold, 3 through 7 underlined in the <i>nroff</i> formatter and italic in <i>troff</i> ) |
| HP          | Heading point size list {4.2.2.4.3}<br>Up to seven codes for heading levels 1 through 7                                                                                                                  |
| Le          | Title for LIST OF EQUATIONS {7.6}                                                                                                                                                                        |
| Lf          | Title for LIST OF FIGURES {7.6}                                                                                                                                                                          |
| Lt          | Title for LIST OF TABLES {7.6}                                                                                                                                                                           |
| Lx          | Title for LIST OF EXHIBITS {7.6}                                                                                                                                                                         |
| RE          | SCCS Release and Level of MM {12.3}<br>Release.Level (e.g., 10.129)                                                                                                                                      |
| Rf          | Reference numberer {11.1}                                                                                                                                                                                |
| Rp          | Title for references {11.4}                                                                                                                                                                              |
| Tm          | Trademark string {3.9} .<br>Places the letters "TM" one-half line above the text that it follows<br>Seven accent strings are also available {12.10}.                                                     |

**Note 1:** If the released-paper style is used, then, in addition to the above strings, certain BTL location codes are defined as strings; these location strings are needed only until the .MT macro is called {6.7} . Currently, the following are recognized:

AK, AL, ALF, CB, CH, CP, DR, FJ, HL, HO, HOH, HP, IH, IN, INH, IW, MH, MV, PY, RD, RR, WB, WH, and WV.

**Note 2:** Paragraph 1.5 has notes on setting and referencing strings.



TABLE 4.C  
NUMBER REGISTER NAMES SUMMARY

| REGISTER            | DESCRIPTION {PARAGRAPH}                                                                                                                                                                                                 |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| A*†                 | Handles preprinted forms and Bell System logo {2.4}<br>0, [0:2]                                                                                                                                                         |
| Au                  | Inhibits printing author's location, department, room, and extension in "from" portion of a memorandum {6.3}<br>1, [0:1]                                                                                                |
| C*†                 | Copy type {2.4}<br>Original, Draft, etc.<br>0 (Original), [0:4]                                                                                                                                                         |
| Cl                  | Contents level {4.4}<br>Level of headings saved for table of contents<br>2, [0:7]                                                                                                                                       |
| Cp                  | Placement of list of figures, etc. {10.1}<br>1 (on separate pages), [0:1]                                                                                                                                               |
| D*†                 | Debug flag {2.4}<br>0, [0:1]                                                                                                                                                                                            |
| De                  | Display eject register for floating displays {7.2}<br>0, [0:1]                                                                                                                                                          |
| Df                  | Display format register for floating displays {7.2}<br>5, [0:5]                                                                                                                                                         |
| Ds                  | Static display pre- and post-space {7.1}<br>1, [0:1]                                                                                                                                                                    |
| E*†                 | Controls font of the Subject/Date/From fields {2.4}<br>1 (nroff) 0 (troff), [0:1]                                                                                                                                       |
| Ec                  | Equation counter, used by .EC macro {7.5}<br>0, [0:?], incremented by one for each .EC call.                                                                                                                            |
| Ej                  | Page-ejection flag for headings {4.2.2.1}<br>0 (no eject), [0:7]                                                                                                                                                        |
| Eq                  | Equation label placement {7.4}<br>0 (right-adjusted), [0:1]                                                                                                                                                             |
| Ex                  | Exhibit counter, used by .EX macro {7.5}<br>0, [0:?], incremented by one for each .EX call.                                                                                                                             |
| Fg                  | Figure counter, used by .FG macro {7.5}<br>0, [0:?], incremented by one for each .FG call.                                                                                                                              |
| Fs                  | Footnote space (i.e., spacing between footnotes) {8.4}<br>1, [0:?]                                                                                                                                                      |
| H1<br>through<br>H7 | Heading counters for levels 1 through 7 {4.2.2.5}<br>0, [0:?], incremented by .H of corresponding level or .HU if at level given by register Hu. H2 through H7 are reset to 0 by any heading at a lower-numbered level. |

\*†See notes at end of table.



TABLE 4.C (Contd)

## NUMBER REGISTER NAMES SUMMARY

| REGISTER | DESCRIPTION {PARAGRAPH}                                                                                                                                                                                                                                |
|----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Hb       | Heading break level (after .H and .HU) {4.2.2.2}2, [0:7]                                                                                                                                                                                               |
| Hc       | Heading centering level for .H and .HU {4.2.2.3}<br>0 (no centered headings), [0:7]                                                                                                                                                                    |
| Hi       | Heading temporary indent (after .H and .HU) {4.2.2.2}<br>1 (indent as paragraph), [0:2]                                                                                                                                                                |
| Hs       | Heading space level (after .H and .HU) {4.2.2.2}<br>2 (space only after .H 1 and .H 2), [0:7]                                                                                                                                                          |
| Ht       | Heading type {4.2.2.5}<br>For .H: single or concatenated numbers<br>0 (concatenated numbers: 1.1.1, etc.), [0:1]                                                                                                                                       |
| Hu       | Heading level for unnumbered heading (.HU) {4.3}<br>2 (.HU at the same level as .H 2), [0:7]                                                                                                                                                           |
| Hy       | Hyphenation control for body of document {3.4}<br>0 (automatic hyphenation off), [0:1]                                                                                                                                                                 |
| L*†      | Length of page {2.4}<br>66, [20:?] (11i, [2i:?] in <b>troff</b> formatter)<br>For <b>nroff</b> formatter, these values are unscaled numbers<br>representing lines or character positions; for <b>troff</b> formatter,<br>these values must be scaled.. |
| Le       | List of equations {7.6}<br>0 (list not produced) [0:1]                                                                                                                                                                                                 |
| Lf       | List of figures {7.6}<br>1 (list produced) [0:1]                                                                                                                                                                                                       |
| Li       | List indent {5.1.1.1}<br>6 ( <b>nroff</b> ) 5 ( <b>troff</b> ), [0:?]                                                                                                                                                                                  |
| Ls       | List spacing between items by level {5.1.1.1}<br>6 (spacing between all levels) [0:6]                                                                                                                                                                  |
| Lt       | List of tables {7.6}<br>1 (list produced) [0:1]                                                                                                                                                                                                        |
| Lx       | List of exhibits {7.6}<br>1 (list produced) [0:1]                                                                                                                                                                                                      |
| N*†      | Numbering style {2.4}<br>0, [0:5]                                                                                                                                                                                                                      |
| Np       | Numbering style for paragraphs {4.1}<br>0 (unnumbered) [0:1]                                                                                                                                                                                           |

\*†See notes at end of table.



TABLE 4.C (Contd)

## NUMBER REGISTER NAMES SUMMARY

| REGISTER | DESCRIPTION {PARAGRAPH}                                                                                                                                                                                                                           |
|----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| O*†      | Offset of page {2.4}<br>.75i, [0:?] (0.5i, [0i:?] in <b>troff</b> formatter)<br>For <b>nroff</b> formatter, these values are unscaled numbers representing lines or character positions; for <b>troff</b> formatter, these values must be scaled. |
| Oc       | Table of contents page numbering style {10.1}<br>0 (lowercase Roman), [0:1]                                                                                                                                                                       |
| Of       | Figure caption style {7.5}<br>0 (period separator), [0:1]                                                                                                                                                                                         |
| P†       | Page number manager by MM {2.4}<br>0, [0:?]                                                                                                                                                                                                       |
| Pi       | Paragraph indent {4.1}<br>5 ( <b>nroff</b> ) 3 ( <b>troff</b> ), [0:?]                                                                                                                                                                            |
| Ps       | Paragraph spacing {4.1}<br>1 (one blank space between paragraphs), [0:?]                                                                                                                                                                          |
| Pt       | Paragraph type {4.1}<br>0 (paragraphs always left justified), [0:2]                                                                                                                                                                               |
| Pv       | "PRIVATE" header {9.10}<br>0 (not printed), [0:2]                                                                                                                                                                                                 |
| Rf       | Reference counter, used by .RS macro {11.1}<br>0, [0:?], incremented by one for each .RS call.                                                                                                                                                    |
| S*†      | The <b>troff</b> formatter default point size {2.4}<br>10, [6:36]                                                                                                                                                                                 |
| Si       | Standard indent for displays {7.1}<br>5 ( <b>nroff</b> ) 3 ( <b>troff</b> ), [0:?]                                                                                                                                                                |
| T*†      | Type of <b>nroff</b> output device {2.4}<br>0, [0:2]                                                                                                                                                                                              |
| Tb       | Table counter, used by .TB macro {7.5}<br>0, [0:?], incremented by one for each .TB call.                                                                                                                                                         |
| U*†      | Underlining style ( <b>nroff</b> ) for .H and .HU {2.4}<br>0 (continuous underline when possible), [0:1]                                                                                                                                          |
| W*†      | Width of page (line and title length) {2.4}<br>6i, [10:1365] (6i, [2i:7.54i] in the <b>troff</b> formatter)                                                                                                                                       |

\*An asterisk attached to a register name indicates that this register can be set only from the command line or before the MM macro definitions are read by the formatter {2.4, 2.5}.

†Paragraph 1.5 has notes on setting and referencing registers. Any register having a single-character name can be set from the command line.



TABLE 4.D

## ERROR MESSAGES

| ERROR MESSAGE                                                                                                                                                                          | DESCRIPTION                                                                                                                                              |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>MM Error Messages</b>                                                                                                                                                               |                                                                                                                                                          |
| An MM error message has a standard part followed by a variable part. The standard part has the form:<br><br>ERROR:( <i>filename</i> )input line <i>n</i>                               |                                                                                                                                                          |
| Variable parts consist of a descriptive message usually beginning with a macro name. They are listed below in alphabetical order by macro name, each with a more complete explanation. |                                                                                                                                                          |
| Check TL, AU, AS, AE, MT sequence                                                                                                                                                      | The correct order of macros at the start of a memorandum is shown in {6.1}. Something has disturbed this order.                                          |
| Check TL, AU, AS, AE, NS, NE, MT sequence                                                                                                                                              | The correct order of macros at the start of a memorandum is shown in {6.1}. Something has disturbed this order. Occurs if the .AS 2 {6.5}macro was used. |
| CS:cover sheet too long                                                                                                                                                                | Text of the cover sheet is too long to fit on one page. The abstract should be reduced or the indent of the abstract should be decreased {6.5}.          |
| DE:no DS or DF active                                                                                                                                                                  | A .DE macro has been encountered, but there has not been a previous .DS or .DF macro to match it.                                                        |
| DF:illegal inside TL or AS                                                                                                                                                             | Displays are not allowed in the title or abstract.                                                                                                       |
| DF:missing DE                                                                                                                                                                          | A .DF macro occurs within a display, i.e., a .DE macro has been omitted or mistyped.                                                                     |
| DF:missing FE                                                                                                                                                                          | A display starts inside a footnote. The likely cause is the omission (or misspelling) of a .FE macro to end a previous footnote.                         |
| DF:too many displays                                                                                                                                                                   | More than 26 floating displays are active at once, i.e., have been accumulated but not yet output.                                                       |
| DS:illegal inside TL or AS                                                                                                                                                             | Displays are not allowed in the title or abstract.                                                                                                       |
| DS:missing DE                                                                                                                                                                          | A .DS macro occurs within a display, i.e., a .DE has been omitted or mistyped.                                                                           |
| DS:missing FE                                                                                                                                                                          | A display starts inside a footnote. The likely cause is the omission (or misspelling) of a .FE to end a previous footnote.                               |
| FE:no FS active                                                                                                                                                                        | A .FE macro has been encountered with no previous .FS to match it.                                                                                       |
| FS:missing DE                                                                                                                                                                          | A footnote starts inside a display, i.e., a .DS or .DF occurs without a matching .DE.                                                                    |
| FS:missing FE                                                                                                                                                                          | A previous .FS macro was not matched by a closing .FE, i.e., an attempt is being made to begin a footnote inside another one.                            |
| H:bad arg: <i>value</i>                                                                                                                                                                | The first argument to the .H macro must be a single digit from one to seven, but <i>value</i> has been supplied instead.                                 |
| H:missing arg                                                                                                                                                                          | The .H macro needs at least one argument.                                                                                                                |
| H:missing DE                                                                                                                                                                           | A heading macro (.H or .HU) occurs inside a display.                                                                                                     |



TABLE 4.D (Contd)

## ERROR MESSAGES

| ERROR MESSAGE                                                                                                                                                                                                                                             | DESCRIPTION                                                                                                                                                                                                                                                                                                                                                                                                                        |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| H:missing FE                                                                                                                                                                                                                                              | A heading macro (.H or .HU) occurs inside a footnote.                                                                                                                                                                                                                                                                                                                                                                              |
| HU:missing arg                                                                                                                                                                                                                                            | The .HU macro needs one argument.                                                                                                                                                                                                                                                                                                                                                                                                  |
| LB:missing arg(s)                                                                                                                                                                                                                                         | The .LB macro requires at least four arguments.                                                                                                                                                                                                                                                                                                                                                                                    |
| LB:too many nested lists                                                                                                                                                                                                                                  | Another list was started when there were already six active lists.                                                                                                                                                                                                                                                                                                                                                                 |
| LE:mismatched                                                                                                                                                                                                                                             | The .LE macro has occurred without a previous .LB or other list-initialization macro {5.1.1} . Although this is not a fatal error, the message is issued because there almost certainly exists some problem in the preceding text.                                                                                                                                                                                                 |
| LI:no lists active                                                                                                                                                                                                                                        | The .LI macro occurred without a preceding list-initialization macro. The latter has probably been omitted or has been separated from the .LI by an intervening .H or .HU.                                                                                                                                                                                                                                                         |
| ML:missing arg                                                                                                                                                                                                                                            | The .ML macro requires at least one argument.                                                                                                                                                                                                                                                                                                                                                                                      |
| ND:missing arg                                                                                                                                                                                                                                            | The .ND macro requires one argument.                                                                                                                                                                                                                                                                                                                                                                                               |
| RF:no RS active                                                                                                                                                                                                                                           | The .RF macro has been encountered with no previous .RS to match it.                                                                                                                                                                                                                                                                                                                                                               |
| RP:missing RF                                                                                                                                                                                                                                             | A previous .RS macro was not matched by a closing .RF.                                                                                                                                                                                                                                                                                                                                                                             |
| RS:missing RF                                                                                                                                                                                                                                             | A previous .RS macro was not matched by a closing .RF.                                                                                                                                                                                                                                                                                                                                                                             |
| S:bad arg:value                                                                                                                                                                                                                                           | The incorrect argument <i>value</i> has been given for the .S macro {12.9} .                                                                                                                                                                                                                                                                                                                                                       |
| SA:bad arg:value                                                                                                                                                                                                                                          | The argument to the .SA macro (if any) must be either 0 or 1. the incorrect argument is shown as <i>value</i> .                                                                                                                                                                                                                                                                                                                    |
| SG:missing DE                                                                                                                                                                                                                                             | The .SG macro occurred inside a display.                                                                                                                                                                                                                                                                                                                                                                                           |
| SG:missing FE                                                                                                                                                                                                                                             | The .SG macro occurred inside a footnote.                                                                                                                                                                                                                                                                                                                                                                                          |
| SG:no authors                                                                                                                                                                                                                                             | The .SG macro occurred without any previous .AU macro(s).                                                                                                                                                                                                                                                                                                                                                                          |
| VL:missing arg                                                                                                                                                                                                                                            | The .VL macro requires at least one argument.                                                                                                                                                                                                                                                                                                                                                                                      |
| WC:unknown option                                                                                                                                                                                                                                         | An incorrect argument has been given to the .WC macro {12.4} .                                                                                                                                                                                                                                                                                                                                                                     |
| <b>Formatter Error Messages</b><br>Most messages issued by the formatter are self-explanatory. Those error messages over which the user has some control are listed below. Any other error messages should be reported to the local system support group. |                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Cannot do ev                                                                                                                                                                                                                                              | Caused by: <ol style="list-style-type: none"> <li>setting a page width that is negative or extremely short</li> <li>setting a page length that is negative or extremely short</li> <li>reprocessing a macro package (e.g., performing a .so request on a macro package that was already requested on the command line)</li> <li>requesting the troff formatter (an option on a document that is longer than ten pages).</li> </ol> |



TABLE 4.D (Contd)

## ERROR MESSAGES

| ERROR MESSAGE                  | DESCRIPTION                                                                                                                                                                                                                                                                                                  |
|--------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Cannot execute <i>filename</i> | Given by the <i>!</i> request if the <i>filename</i> is not found.                                                                                                                                                                                                                                           |
| Cannot open <i>filename</i>    | Indicates one of the files in the list of files to be processed cannot be opened.                                                                                                                                                                                                                            |
| Exception word list full       | Indicates too many words have been specified in the hyphenation exception list (via <i>.hw</i> requests).                                                                                                                                                                                                    |
| Line overflow                  | Indicates output line being generated was too long for the formatter line buffer capacity. The excess was discarded. Likely causes for this message are very long lines or words generated through the misuse of <i>\c</i> of the <i>.cu</i> request or very long equations produced by <i>eqn(1)/neqn</i> . |
| Nonexistent font type          | Indicates a request has been made to mount an unknown font.                                                                                                                                                                                                                                                  |
| Nonexistent macro file         | Indicates the requested macro package does not exist.                                                                                                                                                                                                                                                        |
| Nonexistent terminal type      | Indicates the terminal options refer to an unknown terminal type.                                                                                                                                                                                                                                            |
| Out of temp file space         | Indicates additional temporary space for macro definitions, diversions, etc. cannot be allocated. This message often occurs because of unclosed diversions (missing <i>.FE</i> or <i>.DE</i> ), unclosed macro definitions (e.g., missing <i>".."</i> ), or a huge table of contents.                        |
| Too many page numbers          | Indicates the list of pages specified to the <i>-o</i> formatter option is too long.                                                                                                                                                                                                                         |
| Too many number registers      | Indicates the pool of number register names is full. Unneeded registers can be deleted by using the <i>.rr</i> request.                                                                                                                                                                                      |
| Too many string/macro names    | Indicates the pool of string and macro names is full. Unneeded strings and macros can be deleted using the <i>.rm</i> request.                                                                                                                                                                               |
| Word overflow                  | Indicates a word being generated exceeds the formatter word buffer capacity. Excess characters were discarded. Likely causes for this message are very long lines, words generated through the misuse of <i>\c</i> of the <i>.cu</i> request, or very long equations produced by <i>eqn(1)/neqn</i> .        |







## V. VIEWGRAPHS AND SLIDES MACROS

### 1. Introduction

This section describes a package of UNIX operating system `troff(1)`<sup>1</sup> formatter macros called MV designed for typesetting viewgraphs and slides. It is assumed that the reader has a basic knowledge of the UNIX operating system, the text editor `ed(1)`, and the `troff` formatter.

With the MV macros, viewgraphs can be prepared in a variety of dimensions, as well as 35mm slides and 2x2 "super-slides". These transparencies can be made in a variety of styles, in different fonts, with oversize titles, and with highlighted subordination levels. Because text from which the foils are typeset is stored on the UNIX operating system, the contents of a foil can be readily changed to include new data or can be incorporated into a new presentation. Text of the foils can be passed through `spell(1)`, or preprocessed by `eqn(1)`, `tbl(1)`, `cw(1)`, etc.

It is not possible to include artwork, graphics, or multicolored text in foils made with this macro package except by manual cut-and-paste methods.

### 2. Examples

Before explaining the macros in detail, the formatting process is illustrated with some examples.

#### 2.1 Trivial Example

The following text file is given the file name of *trivial*:

```
.Sw
Six stages of a project:
.B
wild enthusiasm
.B
disillusionment
.B
total confusion
.B
search for the guilty
.B
punishment of the innocent
.B
promotion of the non-participants
```

The `.Sw` is a foil-start macro and is defined in paragraph 3.1. The following UNIX operating system command generates the viewgraph illustrated in Fig. 5.1:

```
mvt trivial
```

<sup>1</sup> The notation `name (N)` indicates entry `name` in Section `N` of the User's Guide—Unix Operating System.



## 2.2 Less Trivial Example

The foil that results from typesetting the following input is illustrated in Fig. 5.2.<sup>2</sup>

```
.Vw 2 " Less Trivial " " June 29, 1980 "
.T " What the Walrus Said "
"The time has come," the Walrus said,
.BR
"To talk of many things:
.I .5
.B
Of shoes\ (em and ships\ (em and sealing wax\ (em
.B
Of cabbages\ (em and kings\ (em
.B
And why the sea is boiling hot\ (em
.B
And whether pigs have wings."
```

The .Vw (paragraph 3.1) is another foil-start macro. Other macros (.T, .BR, and .I) in this example will be explained later.

## 2.3 Other Examples

Inputs that generate foils of Fig. 5.3 through 5.7 are shown below. These foils illustrate the effect of macros that are discussed in the following paragraphs.

The input for Fig. 5.3 is:

```
.Vh 3 " Levels & Marks "
.T " Foil Levels & Level Marks "
This is the .A (left margin) level;
.B
this is the .B level,
.B
as is this;
.C
this is the .C level,
.C
as is this;
.D
and this is the .D level,
.D
as is this.
.A
The large bullet, the dash, and the small
bullet are the default "marks" for
levels .B, .C, and .D, respectively.
However, these three levels can also
be marked arbitrarily:
.B B.
```

<sup>2</sup> The input string \ (em is the troff formatter name for the "em dash" (long dash).



Like this (this is the .B level);  
 .C 3.  
 like this (this is the .C level);  
 .D d.  
 like this (this is the .D level), or  
 .D iv.  
 like this, or even  
 \&.D \(\rh^\(\bu +4  
 like this.  
 .A  
 The .A level cannot be marked.  
 .B  
 An arbitrary number of lines of text  
 can be included in any item at any level;  
 the text will be filled, but neither adjusted  
 nor hyphenated, just like this .B level item.

The input for Fig. 5.4 is:

.DF 1 R  
 .VS 4 Complex  
 .T " Of Bits & Bytes & Words "  
 .S -4  
 .I 3 A x  
 .ft I  
 But let your communication be, Yea, yea;  
 Nay, nay: for whatsoever is more than these  
 cometh of evil.\*  
 .ft  
 .I +1 a nospace  
 Matthew 5:37  
 .BR  
 .S  
 .I 0 .A  
 Binary notation has been around for a  
 .S +6  
 long  
 .S  
 time.  
 .B  
 The above verse tells us to use:  
 .C 1)  
 binary notation,  
 .ft I  
 and  
 .ft  
 .C 2)  
 redundancy  
 .D \(\rh  
 (in communicating)  
 .B  
 Binary notation is  
 .U not  
 suited for human use, above verse to



the contrary notwithstanding.

.SP

.S -2

.TS

box ;

c l c l c l c

l l c l c l c .

System 0 Bits / Byte 0 Bytes / Word 0 Bits / Word

IBM 7090/94 06 06 036

IBM 360/370 08 04 032

PDP 11/70 08 02 016

.TE

.S

.S -4

.U -----

.BR

\* The use of this verse in this context  
is plagiarized from C. Shannon.

.S

The input for Fig. 5.5 is:

.de CW

.I .5 a

.NF

..

.de CN

.FI

.I 0 a

..

.DF 1 R 2 I 3 CW

.VS 5 "CW & EQN"

.EQ

gsize 18

.EN

.S 100 5.5

Input:

.CW

.EQ

sum from k=1 to inf m sup k-1

$\sim = \sim 1 \text{ over } 1-m$

.EN

.CN

Output:

.I 2 a

.EQ

sum from k=1 to inf m sup k-1

$\sim = \sim 1 \text{ over } 1-m$

.EN

.I 0 a

Input:

.CW

The equation \$ f(t)  $\sim = \sim 2 \pi$



int sin ( omega t ) dt \$  
is used here in running text,  
rather than being displayed.

.CN

Output:

.I .5 a

.EQ

delim \$\$

.EN

.AD

The equation \$ f(t) = \int\_0^{2\pi} \sin(\omega t) dt \$

is used here in running text,

rather than being displayed.

.EQ

delim off

gsize 10

.EN

The input for Fig. 5.6 is:

.VS 6 " The Works: Input "

Input:

.S -4

.CW

.TS

center doublebox ;

Cip+4 ! Cip+4 S S

^ ! L L L

^ ! C I C I C

^ ! C I C I C

Li ! C I C I N .

Users@Hardware

@\_@\_@\_

@UNIX\\* (Tm@Model@Serial

@System@\^@Number

=

OS Dev.@A@VAX@54

SGS DEV.@B@11/70@3275

Low-End@C@11/23@221

- And now ...@T{

.NA

Some filled text and an equation:

T}@T{

\$ zeta (s) = prod

from K=1 to inf k sup -s \$

.AD

T}@1.2

.TE

.CN

The input for Fig. 5.7 is:

.VS 7 " The Works: Output "



```
.EQ
delim $$
gsize 14
.EN
Output:
.I 0 a
.SP
.TS
center doublebox ;
Cip+4 | Cip+4 S S
^ | L L L
^ | C | C | C
^ | C | C | C
Li | C | C | N .
UsersOHardware
O _ O _ O _
O UNIX \ *(TmOModelOSerial
O SystemO \ ^ONumber
=
OS Dev.OAOVAXO54
SGS Dev.OBO11/70O3275
Low-EndOCO11/23O221
-
And now ...OT{
.NA
Some filled text and an equation:
T)OT{
$ zeta (s) = prod
from k=1 to inf k sup -s $
.AD
T{O1.2
.TE
.EQ
delim off
gsize 10
.EN
```

### 3. Macros

The following is an explanation of the MV macros which are summarized in **mv(7)** of the User's Guide—UNIX Operating System.

#### 3.1 Foil-Start Macros

Each foil must start with a foil-start macro. There are nine foil-start macros for generating nine different-sized foils; the names (and the corresponding mounting-frame sizes) of these macros are shown in Table 5.A.

The naming convention for these nine macros is that the first character of the name (V or S) distinguishes between viewgraphs and slides, while the second character indicates whether the foil is square (S), small wide (w), small high (h), big wide (W), or big high (H). Slides are thinner than the corresponding viewgraphs; therefore the ratio of the longer dimension to the shorter one is larger for slides than for viewgraphs. As a result, slide foils can be used for viewgraphs, but not vice versa. On the other hand, viewgraphs can accommodate a bit more text.

**Note:** The .VW and .SW macros produce foils that are 7x5.4 inches because commonly available typesetter paper is less than 9 inches wide. These foils must be enlarged by a factor of 9/7 before they can be used as 9-inch wide by 7-inch high viewgraphs.



Each foil-start macro causes the previous foil (if any) to be terminated, foil separators to be produced, and certain heading information to be generated. The default heading information consists of three lines of right-justified data:

- The current date in the form *mo/dy/yr*
- BTL
- FOIL *n*

where *n* is the sequence number in the current "run". As explained below, this heading information is replaced by the three arguments of the foil-start macro if those arguments are given.

The actual projection area is marked by "cross hairs" (plus signs) that fit into the corners of the viewgraph mount. This is an aid in positioning the foil for mounting.

All foils other than the square (.VS) foil also have a set of horizontal and vertical "crop marks". These indicate how much of the foil will be seen if it is made into a slide, rather than into a viewgraph.

Default heading information can be changed by specifying three optional arguments to the foil-start macro. Square brackets ([ ]) indicate that the argument they enclose is optional.

.XX [ *n* ] [ *id* ] [ *date* ]

where:

- XX stands for one of the nine foil-start macros
- *n* is the foil identifier (typically a number)
- *id* is other identifying information (typically the initials of the person creating the foil)
- *date* is usually the date.

The resulting heading information consists of three lines of right-justified text:

- *id*
- *date*
- FOIL *n*.

If *date* and *id* are omitted on a foil-start macro, then the corresponding values (if any) from the previous foil-start macro are used.

### 3.2 Level Macros

The MV macros provide four levels of indentation, called .A, .B, .C, and .D. Each of these level macros causes the text that follows it to be placed at the corresponding level of indentation.

The amount of vertical spacing done by each level macro can be changed with the .DV macro (paragraph 3.7). Figure 5.3 is devoted to the level macros.



### 3.2.1 The .A Level

.A [ x ]

The leftmost level (left margin) is obtained by the .A macro. The .A level is automatically invoked by each of the foil-start macros. Each .A macro spaces a half-line from the preceding text, unless the *x* argument is specified (*x* can be any character or string of characters); *x* suppresses the spacing.

The .A macro does not generate a mark of any sort; it is the "left-margin" macro. Repeated .A calls are ignored, but each successive call of any of the other three level macros generates the corresponding mark.

The .A macro can also be invoked through the .I macro (paragraph 3.4).

### 3.2.2 The .B Level

.B [ mark [ size ] ]

The .B level items are marked by a bullet (in slightly reduced point size). The text that follows the .B macro is spaced one half-line from the preceding text.

The .B level *mark* may be changed by specifying the desired character string<sup>3</sup> as the first argument. Without the second argument (*size*), the point size of the *mark* is not reduced. Thus, the following will produce a numbered list:

.VS

This is a list of things:

.B 1.

This is thing number 1.

.B 2.

This is thing number 2.

.B 3.

This is the third and last thing on this foil.

It is possible to change the point size of the *mark* with the second argument (*size*). If given, it specifies the desired point-size change. An unsigned or positive (+) argument is taken as an increment; a negative (-) argument is a decrement. An argument greater than 99 causes the *mark* to be reduced in size just as if it were the default *mark*, namely, the bullet. After the *mark* is printed, the previous point size is restored. All these point-size changes are completely invisible to the user.

### 3.2.3 The .C Level

.C [ mark [ size ] ]

The .C level is like the .B level except that it is indented farther to the right and the default *mark* is a long dash (\(em)) in a slightly reduced point size.

### 3.2.4 The .D Level

.D [ mark [ size ] ]

The .D level is indented farther to the right than the .C level and does not space from the previous text. It causes the following text to start on a new line. In other words, it causes a break (paragraph 3.10). Otherwise, it behaves like the .B and .C levels. The .D level default *mark* is a bullet smaller than that used for the .B level.

<sup>3</sup> All character-string arguments that contain spaces must be quoted ("...").



### 3.3 Titles

#### .T string

The .T macro creates a centered title from its argument (*string*). The argument must be enclosed within double quotes ( " ... " ) if it contains spaces. The size of the title is four points larger than prevailing point size. Any indentation established by the .I macro (paragraph 3.4) has no effect on titles; they are always centered within the foil horizontal dimension.

Figures 5.2, 5.3, and 5.4 illustrate the .T macro.

### 3.4 Global Indents

#### .I [ indent ] [ a [ x ] ]

The entire text (except titles) of the foil may be shifted right or left by the .I macro. The first argument (*indent*) is the amount of indentation that is to be used to establish a new left margin. This argument may be signed positive or negative, indicating right or left movement from the current margin. If unsigned, the argument specifies the new margin, relative to the initial default margin. If the argument is not dimensioned, it is assumed to be in inches (see [3,8] for legal troff formatter units). If the argument is null or omitted, 0i is assumed causing the margin to revert to the initial default margin.

If a second argument is specified, the .I macro calls the .A macro (paragraph 3.2.1) before exiting. The third argument, if present, is passed to the .A macro.

Figures 5.2, 5.4, 5.5, and 5.7 illustrate the .I macro.

### 3.5 Point Sizes and Line Lengths

#### .S [ ps ] [ ll ]

Each foil-start macro begins the foil with an appropriate default point size<sup>4</sup> and line length. Prevailing point size and line length may be changed by invoking the .S macro. If the *ps* argument is null, the previous point size is restored. If *ps* is signed negative, the point size is decremented by the specified amount. If *ps* is signed positive, it is used as an increment; and if *ps* is unsigned, it is used as the new point size. If *ps* is greater than 99, the initial default point size is restored (Table 5.B). Vertical spacing is always 1.25 times the current point size.

The second argument (*ll*), if given, specifies line length. It may be dimensioned. If it is not dimensioned and less than 10, it is taken as inches. If it is not dimensioned and greater than or equal to 10, it is taken as troff formatter units (1/432nds of an inch) (paragraph 7.3).

Figures 5.4, 5.5, and 5.6 illustrate the .S macro.

### 3.6 Default Fonts

#### .DF n font [ n font ... ]

The MV macros assume that the Helvetica Regular (also known as Geneva) font, mounted in position 1, is the default font. Additional fonts can be mounted and the default font can be changed. The .DF macro informs the troff formatter that *font* is in position *n*. The first-named font is the default font. Up to four pairs of arguments may be specified.

<sup>4</sup> Default point sizes for each type of foil and corresponding maximum number of lines are given in Table 5.B.



The .DF macro must immediately precede a foil-start macro; the initial setting is equivalent to

```
.DF 1 H 2 I 3 B 4 S
```

Figures 5.4 and 5.5 illustrate the .DF macro.

### 3.7 Default Vertical Space

```
.DV [ a ] [ b ] [ c ] [ d ]
```

The default vertical space macro (.DV) allows changing the vertical spacing done by each of the four level macros (paragraph 3.2). The first argument (*a*) is the spacing for the .A macro, *b* is for the .B macro, *c* is for the .C macro, and *d* is for the .D macro. All non-null arguments must be dimensioned. Null arguments leave the corresponding spacing unaffected. The initial setting is equivalent to

```
.DV .5v .5v .5v 0v
```

### 3.8 Underlining

```
.U string1 [ string2 ]
```

The underline macro (.U) takes one or two arguments. The first argument (*string1*) is the string of characters to be underlined. The second argument (*string2*), if present, is not underlined but concatenated to the first argument. For example:

```
.U phototypesetter
```

produces

phototypesetter

while

```
.U under line
```

produces

underline

Figure 5.4 illustrates the .U macro.

### 3.9 Synonyms

The MV macro package recognizes the .AD, .BR, .CE, .FI, .HY, .NA, .NF, .NH, .NX, .SO, .SP, .TA, and .TI uppercase text synonyms for the corresponding lowercase troff formatter requests. The *NROFF and TROFF User's Manual* found in part 3 of this document contains definitions of these requests.

### 3.10 Breaks

The .S, .DF, .DV, and .U macros do not cause a break. The .I macro causes a break only if it is invoked with more than one argument. All other MV macros always cause a break. The troff formatter synonyms (paragraph 3.9) .AD, .BR, .CE, .FI, .NA, .NF, .SP, and .TI also cause a break.



### 3.11 Text Filling, Adjusting, and Hyphenation

By default, the MV macros fill, but neither adjust nor hyphenate text. This is an aesthetic judgement that seems correct for foils. These defaults can, of course, be changed by using the .AD, .FI, .HY, .NA, .NF, and .NH macros (paragraph 3.9).

## 4. The troff Preprocessors

It is possible to use the various **troff** formatter preprocessors to typeset foils that require more powerful formatting capabilities.

### 4.1 Tables

The **tbl(1)** program can be used to set up columns of data within a viewgraph or slide. The .TS and .TE macros are not defined in the MV macro package, but are merely flags to **tbl(1)**. The *Table Formatting Program* found in part 3 of this document describes the macros used for generating tables. Figures 5.4 and 5.7 illustrate the **tbl** program use.

### 4.2 Mathematical Expressions

The **eqn(1)** program can be used to typeset mathematical expressions and formulas on foils provided care is taken to specify proper fonts and point sizes. The *Mathematics Typesetting Program* found in part 3 of this document describes the macros used for processing equations. The .EQ and .EN macros are not defined in the MV macro package. Figures 5.5 and 5.7 illustrate the **eqn** program.

### 4.3 Constant-Width Program Examples

The constant-width font simulates computer-terminal and line-printer output and can be sometimes effective in presenting computer-related topics. The **cw(1)** program, as well as Figures 5.5 and 5.6 illustrate the preprocessor.

## 5. The Finished Product

### 5.1 Phototypesetter Output

```
mvt [ options ] file ...
```

Typeset output is obtained via the **mvt** command where the argument *file* contains text and macro invocations for the foils, and *options* can be one or more of the following:

- a            preview output on a terminal (other than a Tektronix 4014—paragraph 5.2)
- e            invoke **eqn(1)**
- t            invoke **tbl(1)**
- T*term*       direct output to *term*, where *term* can be one of the following:
  - st          STARE
  - 4014        Tektronix 4014
  - vp          Versatec printer

Using a hyphen (-) in place of *file* causes the **mvt** command to read the standard input (rather than a file), as in the following example using the **cw(1)** preprocessor (paragraph 4.3).



```
cw [ options ] file ... | mvt [ options ] -
```

## 5.2 Output Approximation on a Terminal

```
mvt -a file_name ...
```

An approximation of the typeset output can be obtained by entering the `mvt` command. The resulting output shows the formatted foils except that:

- Point-size changes are not visible
- Font changes cannot be seen
- Titles that are too long appear proper
- All horizontal motions are reduced to one horizontal space to the right
- All vertical motions are reduced to one vertical space down.

For example, it appears that lines of text following a `.B`, `.C`, or `.D` macro do not align properly (even though, in fact, they do).

Although alignment cannot be determined from this approximation, line breaks and the amount of vertical space used by the text can be observed. If the foil is not full, the macro package prints the number of blank lines (in the then current point size) that remain on the foil; if the foil is full, a warning is printed. If the text did overflow the foil, text will be printed after the "cross hairs."

## 5.3 Making Actual Viewgraphs and Slides

Output of the typesetter is so-called "mechanical paper," which is white, opaque photographic paper with black letters. There are several very simple processes (e.g., Thermofax, Bruning) for making transparent foils from opaque paper. Because some of these processes involve heat and because mechanical paper is heat sensitive, one should first make copies of the typesetter output on a good quality office copier and then use these copies for making transparencies.

Getting slides made is a much more complicated photographic process that is best left to professionals. It is possible to make both positive (opaque letters on transparent background) and negative (transparent letters on opaque background) slides, as well as colored-background slides, etc.

## 6. Suggestions For Use

The following suggestions have been derived from experience, from the examination of several other macro packages for making foils, and from some publications that discuss good and bad foil-making practices:

- The most useful foil sizes are `.VS` and `.Vw` (or `.Sw`). This is because most projection screens are either square or wide (wider than they are tall) and also because the resulting foils are smaller, easier to carry, and require no enlargement before use.
- Reducing point size below the default value should be avoided. Default point size for each type of foil (Table 5.B) is the smallest point size that will result in a foil that is legible by an audience of more than a dozen people. If there is more text than fits onto a foil, two or more foils should be used instead of reducing the point size.
- Numerous font changes should be avoided. A foil with more than two typefaces looks cluttered and distracts the viewer.



- Underlined typeset text should be avoided. Even though this package contains a macro for underlining, it should not be used. Underlined typeset text almost always looks bad; instead use a different typeface.
- The Helvetica sans-serif font is thicker and easier to read than the Times Roman serif font normally used for typesetting. On the other hand, the Times Roman font permits more text to be squeezed onto a foil. If it is intended to use italic and/or bold typefaces, either the Helvetica regular, italic, and medium<sup>5</sup>

.DF 1 H 2 HI 3 HM

or the Times Roman regular, italic, and bold

.DF 1 R 2 I 3 B

should be mounted via the .DF macro (paragraph 3.6). Bold typefaces tend to be a bit overwhelming. Choice of fonts is primarily a matter of personal aesthetics. The following table identifies fonts used in the examples of Fig. 5.1 through 5.7.

1, 2, and 3 H (default)

4, 7 R and I

5, 6 R and CW

- The .SP macro can be used to insert a bit of additional white space (for instance, .5v or 1v, where v means "vertical space") at the top of each foil (i.e., increase the top margin).
- Normal uppercase and lowercase text is more legible than uppercase text only.<sup>6</sup> Uppercase and lowercase alphabets have evolved and been used for many years because they result in more legible text. Furthermore, such text is less bulky than uppercase text only, so more information can be put onto a foil without crowding.
- Foils for a presentation should be made as consistent as possible. Changing fonts, typefaces, point sizes, etc., from foil to foil tends to distract the viewer. While it is possible to introduce emphasis and draw the viewer's attention to particular items with such changes, this works only if it is done purposefully and sparingly. Overuse of these techniques is almost always counter-productive.

In summary, the dictum that "the medium is the message" does not apply to foil making. When in doubt:

- Do not change point sizes.
- Do not change fonts or typefaces.
- Do not underline.
- Use many "sparse" foils rather than a few "dense" ones.
- Use fewer words rather than more.
- Use larger point sizes rather than smaller.

<sup>5</sup> Helvetica Medium is really a bold typeface.

<sup>6</sup> The only exception to this rule are foils set in a point size so small that lowercase characters simply cannot be read. This is usually the case for foils produced on a normal typewriter.



- Use larger top and bottom margins rather than smaller.
- Use normal uppercase and lowercase text rather than uppercase text only.

## 7. Warnings

### 7.1 Use of troff Formatter Requests

In general, it is not advisable to intermix arbitrary **troff** formatter requests with the MV macros because this often leads to undesirable (and sometimes astonishing) results. The "safe" requests are ones for which uppercase text synonyms have been defined in the MV package (paragraph 3.9). Other **troff** formatter requests should be used sparingly (if at all) and with care and discipline. Particularly dangerous are requests that affect point size, indentation, page offset, line and title lengths, and vertical spacing between lines. The **.S** and **.I** macros should be used instead (paragraphs 3.5 and 3.4).

### 7.2 Reserved Names

Certain names are used internally by this macro package. In particular, all 2-character names starting with either **)** or **]** are reserved. Names that are the same as names of the MV macros and strings described in this part or names that are the same as **troff** names cannot be used. Furthermore, if any of the preprocessors (Part 4) are used, their reserved names must also be avoided.

### 7.3 Miscellaneous

The **.S** macro changes the point size and vertical spacing immediately, but a line-length change requested with that macro does not take effect until the next-level macro call.

Specifying a third argument to the **.S** macro usually results in a disaster.

The string **Tm** (invoked as **\\*(Tm)** generates a trademark symbol.

The tilde (**~**) is defined by the MV macros as a "nonpaddable" space; that is, the tilde may be used wherever a fixed-size (nonadjustable) space is desired. To override this definition, the following line should be included in the input file:

```
.tr ~
```

## 8. Dimensional Details

For each style of viewgraph Table 5.B shows the default point size; the maximum number of lines of text (at the default point size); and the height, width, and aspect ratio, both nominal and actual.



5/21/82  
BTL  
FOIL 1

Six stages of a project:

- wild enthusiasm
- disillusionment
- total confusion
- search for the guilty
- punishment of the innocent
- promotion of the non-participants

Fig. 5.1 — Trivial Example



June 29, 1980  
Less Trivial  
FOIL 2

## What the Walrus Said

"The time has come," the Walrus said,  
"To talk of many things:

- Of shoes—and ships—and sealing wax—
- Of cabbages—and kings—
- And why the sea is boiling hot—
- And whether pigs have wings."

Fig. 5.2 — Less Trivial Example



June 29, 1980  
Levels & Marks  
FOIL 3

## Foil Levels & Level Marks

This is the .A (left margin) level;

- this is the .B level,
- as is this;
  - this is the .C level,
  - as is this;
    - and this is the .D level,
    - as is this.

The large bullet, the dash, and the small bullet are the default "marks" for levels .B, .C, and .D, respectively. However, these three levels can also be marked arbitrarily:

B. Like this (this is the .B level);

3. like this (this is the .C level);

d. like this (this is the .D level), or

iv. like this, or even

☞ ● like this.

The .A level cannot be marked.

- An arbitrary number of lines of text can be included in any item at any level; the text will be filled, but neither adjusted nor hyphenated, just like this .B level item.

Fig. 5.3 — Example of Foil Levels



June 29, 1980  
Complex  
FOIL 4

## Of Bits & Bytes & Words

*But let your communication be,  
Yea, yea; Nay, nay: for whatsoever  
is more than these cometh of evil.\**

Matthew 5:37

Binary notation has been around for a long time.

- The above verse tells us to use:
  - 1) binary notation, *and*
  - 2) redundancy  
☞ (in communicating)
- Binary notation is not suited for human use, above verse to the contrary notwithstanding.

| System      | Bits/Byte | Bytes/Word | Bits/Word |
|-------------|-----------|------------|-----------|
| IBM 7090/94 | 6         | 6          | 36        |
| IBM 360/370 | 8         | 4          | 32        |
| PDP 11/70   | 8         | 2          | 16        |

\* The use of this verse in this context is plagiarized from C. Shannon.

Fig. 5.4 — Example of Square Foil



June 29, 1980  
The Works: Input  
FOIL 6

Input:

.TS (→ = tab)  
center doublebox ;  
Cip+4 | Cip+4 S S  
^ | L L L  
^ | C | C | C  
^ | C | C | C  
Li | C | C | N .  
Users→Hardware  
→\_→\_→\_  
→UNIX\\*(Tm→Model→Serial  
→System→\^→Number  
=  
OS Dev.→A→VAX→54  
SGS Dev.→B→11/70→3275  
Low-End→C→11/23→221  
-  
And now ...→T{  
.NA  
Some filled text and an equation:  
T}→T{  
\$ zeta (s) = prod  
from k=1 to inf k sup -s \$  
.AD  
T}→1.2  
.TE

Fig. 5.6 — Example of Input of a Table Foil



June 29, 1980  
CW & EQN  
FOIL 5

Input:

```
.EQ
sum from k=1 to inf m sup k-1
==~ 1 over 1-m
.EN
```

Output:

$$\sum_{k=1}^{\infty} m^{k-1} = \frac{1}{1-m}$$

Input:

The equation \$ f(t) ==~ 2 pi  
int sin ( omega t ) dt \$  
is used here in running text,  
rather than being displayed.

Output:

The equation  $f(t) = 2\pi \int \sin(\omega t) dt$  is  
used here in running text, rather than  
being displayed.

Fig. 5.5 — Example of Indent



June 29, 1980  
The Works: Output  
FOIL 7

Output:

| <i>Users</i>       | <i>Hardware</i>                            |                                          |                  |
|--------------------|--------------------------------------------|------------------------------------------|------------------|
|                    | UNIX <sup>TM</sup><br>System               | Model                                    | Serial<br>Number |
| <i>OS Dev.</i>     | A                                          | VAX                                      | 54               |
| <i>SGS Dev.</i>    | B                                          | 11/70                                    | 3275             |
| <i>Low-End</i>     | C                                          | 11/23                                    | 221              |
| <i>And now ...</i> | Some<br>filled text<br>and an<br>equation: | $\zeta(s) = \prod_{k=1}^{\infty} k^{-s}$ | 1.2              |

Fig. 5.7 — Example of Output of a Table Foil



TABLE 5.A  
FOIL—START MACROS

| MACRO NAME | SIZE* AND TYPE                      | BTL FRAME NUMBER†  |
|------------|-------------------------------------|--------------------|
| .VS        | 7×7 viewgraph or<br>2×2 super-slide | E-7351 or E-7351-R |
| .Vw        | 7×5 viewgraph                       | E-7351-B           |
| .Vh        | 5×7 viewgraph                       | E-7351-A           |
| .VW        | 9×7 viewgraph                       | E-8814 or E-9148   |
| .VH        | 7×9 viewgraph                       | E-8814 or E-9148   |
| .Sw        | 7×5 35mm slide                      | E-7351-B           |
| .Sh        | 5×7 35mm slide                      | E-7351-A           |
| .SW        | 9×7 35mm slide                      | E-8814 or E-9148   |
| .SH        | 7×9 35mm slide                      | E-8814 or E-9148   |

\*Size of mounting frame opening (width times height)  
in inches.

†BTL stock item number.

TABLE 5.B  
DEFAULT POINT SIZE, DIMENSIONS, AND ASPECT RATIOS

| MACRO<br>NAME<br>(NOTE 1) | POINT<br>SIZE | MAX.<br>LINES | NOMINAL  |     |     |                    | ACTUAL (TEXT) |     |      |                    |
|---------------------------|---------------|---------------|----------|-----|-----|--------------------|---------------|-----|------|--------------------|
|                           |               |               | W        | H   | AR  | <sup>1</sup><br>AR | W             | H   | AR   | <sup>1</sup><br>AR |
|                           |               |               | (NOTE 2) |     |     |                    | (NOTE 2)      |     |      |                    |
| .VS                       | 18            | 21            | 7        | 7   | 1   | 1                  | 6             | 6.8 | 1.13 | .88                |
| .Vw                       | 14            | 19            | 7        | 5   | .71 | 1.4                | 6             | 4.8 | .8   | 1.25               |
| .Vh                       | 14            | 27            | 5        | 7   | 1.4 | .71                | 4.2           | 6.8 | 1.6  | .62                |
| .VW                       | 14            | 21            | 7        | 5.4 | .77 | 1.3                | 6             | 5.2 | .87  | 1.15               |
| .VH                       | 18            | 28            | 7        | 9   | 1.3 | .77                | 6             | 8.8 | 1.5  | .68                |
| .Sw                       | 14            | 18            | 7        | 4.6 | .67 | 1.5                | 6             | 4.4 | .73  | 1.4                |
| .Sh                       | 14            | 27            | 4.6      | 7   | 1.5 | .67                | 3.8           | 6.8 | 1.8  | .56                |
| .SW                       | 14            | 18            | 7        | 4.6 | .67 | 1.5                | 6             | 4.4 | .73  | 1.4                |
| .SH                       | 18            | 28            | 6        | 9   | 1.5 | .67                | 5             | 8.8 | 1.76 | .57                |

**Note 1:** The .SW (if used as a viewgraph) and .VW foils must be enlarged by a factor of 9/7.

**Note 2:** W—Width in inches.  
H—Height in inches.  
AR—Aspect ratio (H/W).



# UNIX Sytem V

## Init and Getty



Trademarks:

MUNIX, CADMUS  
DEC, PDP  
UNIX

for PCS  
for DEC  
for Bell Laboratories

Copyright 1984 by  
PCS GmbH, Pfälzer-Wald-Strasse 38, D-8000 München 90, tel. (089) 67804-0

The information contained herein is the property of PCS and shall neither be reproduced in whole or in part without PCS's prior written approval nor be implied to grant any license to make, use or sell equipment manufactured herewith.

PCS reserves the right to make changes without notice in the specifications and materials contained herein and shall not be responsible for any damages (including consequential) caused by reliance on the materials presented.



# UNIX<sup>TM</sup> System V Init and Getty

## 1. Introduction

In the UNIX\* system environment, the initial process spawning is controlled and overseen by the first process forked by the UNIX operating system as it comes up at boot time. This process is known as *init*. One of the major jobs of *init* is to fork processes which will become the *getty-login-sh* sequence. This sequence of processes allows users to *login* and takes care of setting up the initial conditions on the outgoing terminal lines so that the speed and the other terminal related states are correct. *Init* and these other processes also keep an accounting file */etc/wtmp* that is available to processes on the system. With these files it is possible to determine the state of each process that *init* has spawned, and if it is a terminal line, who the current user is. One program in particular, *who(1)*, provides a means of examining these files.

This document describes the capabilities of each program used in this new implementation, the databases involved, and how to create and maintain these databases. In addition, the debugging features designed into both *init* and *getty* are described in the event remedial action is required or modifications are attempted.

## 2. Init

*Init* is driven by a database, its previous internal level, its current internal level, and events which cause it to wake up.

### 2.1 The Database: */etc/inittab*

*Init*'s database, kept in the file */etc/inittab*, consists of any number of separate entries, each with the form:

**id:level:type:process**

|       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| id    | The <i>id</i> is a one to four letter identifier which is used by <i>init</i> internally to label entries in its process table. It is also placed in the dynamic record file, <i>/etc/utmp</i> , and the history file, <i>/etc/wtmp</i> . The <i>id</i> should be unique.                                                                                                                                                                                                                                                                                                                            |
| level | The <i>level</i> specifies at which levels <i>init</i> should be concerned with this entry. <i>Level</i> is a string of characters consisting of [0-6a-c]. Anytime that <i>init</i> 's internal level matches a level specified by <i>level</i> , this entry is <i>active</i> . If <i>init</i> 's internal level does not match any of the levels specified, then <i>init</i> makes certain that the process is not running. If the level field is empty it is equivalent to the string "0123456".                                                                                                   |
| type  | The <i>type</i> specifies some further condition required for or by the execution of an entry.<br>off           The entry is not to run even if the levels match.<br>once          The entry is to be run only if <i>init</i> is entering a level. This means if <i>init</i> has been awakened by powerfail or because a child died this entry will not be activated. Only when a user signal requests a change of <i>init</i> 's internal state to a state which is different from its current state, and this new state is one in which this entry should be active, will this entry be activated. |

---

\* UNIX is a Trademark of Bell Telephone Laboratories, Incorporated.



- wait** *Wait* has all the characteristics of *once*, plus it causes *init* to wait until the process spawned dies before reading anymore entries from its database. This allows for initialization actions to be performed and completed before allowing other processes which might be affected to start running. It is common in the OSS environment for shared memory segments to be initialized this way and semaphores to be conditioned.
- respawn** *Respawn* requests that this entry continue to run as long as *init* is running in a level which is in this entry's *level* field. Most processes spawned by *init* fall into this category. All *getty* processes are marked as *respawn*. Whenever *init* detects the death of a process that was marked *respawn*, it spawns a new process to take its place.
- boot** *Boot* entries have the execution behavior of *once* entries. They are started only when *init* is switching to a numeric run state for the first time. Most commonly *boot* entries have an empty *level* string, meaning that no matter which level *init* switches to the first time, the *boot* entry will be run. Should there be a more specific *level* string, for example "01", then the *boot* entry would only be run if *init* switched to either the 0 or 1 run state as its first numeric level.
- bootwait** *Bootwait* entries have the execution behavior of *wait* entries and they, like *boot* entries, are only run as *init* switches to a numeric level for the first time.
- power** *Power* entries act like *once* entries and are activated if *init* receives a SIGPWR signal (19) and is in a state which matches the active states for the entry.
- powerwait** *Powerwait* entries act like *wait* entries and are activated if *init* receives a SIGPWR signal and is in a state which matches the active states for the entry.
- initdefault** *Initdefault* is a non-standard entry in that it does not specify some process to be spawned. Instead it only specifies which level *init* is to go to initially when it is coming up at boot time. This allows the system to be rebooted without an operator having to make entries at the system console if so desired. If there is no *initdefault* entry, then *init* will ask at the system console, */dev/syscon*, for the initial run state. In addition to specifying the numbered states, the single-user state [s] may also be specified.
- process** The *process* field is the action that *init* will ask a *sh* to perform whenever the entry is activated. The string in the *process* field is given a prefix of "exec " so that each entry will only generate one process initially. *Init* then forks and execs

```
sh -c "exec process"
```

This means that the *process* string can take full advantage of all *sh* syntax. The only peculiarities arise from the string "exec ", which was prefixed to the string, and because initially there is no standard input, output, or error output. The addition of "exec " to the string means that if the user wants to have a single entry generate more than one process, for example making a list of the people on the system at the time of a powerfail and mailing it to *root* by the command "who | mail root", it would have to be put in as

```
pf::powerwait:sh -c "who | mail root"
```

to work. If it was put in simply as "who | mail root", it would be executed as "exec who | mail root", and only the *who* process would be created before the *sh* disappeared. The



lack of standard input and output channels must be addressed by explicitly specifying them. An example is the **blog** program that many OSS's run as a *bootwait* entry as the system comes up. Since it requires the operator to supply input, it appears as

```
bl::bootwait:/etc/blog </dev/syscon >/dev/syscon 2> &1  
in /etc/inittab.
```

## 2.2 Levels

A level is one of seven numeric levels, denoted 0, 1, 2, 3, 4, 5, or 6, three temporary levels, denoted a, b, or c, or the single-user level, s. Normally *init* runs in a numeric level. Precisely how a particular level is used depends entirely on the database and the system administrator. The temporary levels allow certain entries to be started on demand without affecting any processes that were started at a particular level. The temporary levels immediately revert to the previous numeric level once all entries in the database have been scanned to see if they should be started at the temporary level. When an entry is started by a switch to a temporary level, it becomes independent of future level changes by *init*, except a change to the single-user level. The only way to kill a process that was started as a respawnable demand process, without going to the single-user level, is to modify the database, declaring the entry to be *off*.

The single-user level is the one level independent of the database. For this reason it is not a level in the normal sense. In the single-user level *init* spawns off a *su* process on the system console, and that is the only process that it maintains while at the single-user level. The single-user level can be entered at two different places in *init*. If it is entered at boot time it allows the operator to look over the file systems without having *init* attempt to do any file I/O, which might cause further problems. *Init* will not attempt to recreate */etc/utmp* or access */etc/wtmp* until after it has left this initial single-user level. If the single-user level is entered at any other time, *init* does do the bookkeeping in the record files.

The system administrator requests *init* to change levels by running a secondary copy of *init* itself. */etc/init* is linked to */bin/telinit*, and it is usually through the *telinit* name that this is accomplished. *Init* can only be run by root or a privileged group. Whenever *init* starts running and finds that its process id is not 1, it assumes that it is a user initiated copy, which is supposed to send a signal to the real *init*. The usage is:

```
telinit {0123456sSqQabc}
```

and the single character argument specifies the signal to be sent to *init*. If the request is to switch to the single-user level, 'S' or 's', then *init* also relinks */dev/syscon* to the terminal originating the request so that it becomes the virtual system console, thus insuring that future messages from *init* will be directed to the terminal where the operator is located. When it does this relinking it also sends a message to */dev/systty*, saying that the console is being relinked to some other terminal so that there is a record of the fact at the physical system console.

## 2.3 Waking Events

There are four events which will wake *init*: boot, a powerfail, death of a child process, or a user signal.

- |                    |                                                                                                                                                                                                                                                                                                                            |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>boot</b>        | <i>Init</i> operates in the <b>boot</b> state until it has entered a numeric state for the first time. It is not possible for <i>init</i> to reenter the boot state a second time. Commands labeled <i>boot</i> and <i>bootwait</i> are executed when changing to a numeric state for the first time, if the levels match. |
| <b>powerfail</b>   | Any time power fails, the operating system sends a SIGPWR signal to all processes. <i>Init</i> will execute commands with types of <i>power</i> and <i>powerfail</i> .                                                                                                                                                     |
| <b>child death</b> | Any time a child process of <i>init</i> dies, <i>init</i> receives a SIGCLD signal (18). The dead child process may be one of two types, a direct decendent of <i>init</i> , or a process whose own parent process died before it did. The parent of a process automatically                                               |



becomes *init*, if its real parent should die before it does. *Init* determines immediately if the defunct process was one of its own children or an *orphan*. If it was one of its own, it performs the necessary bookkeeping on its internal process table to note that the process died. If *init* was busy at the time it received the SIGCLD signal, it then returns to complete whatever action it was performing. If *init* was asleep, it then scans its database to determine if any other actions should be taken, such as respawning the process.

**user signal** *Init* catches all signals that it is possible for a process to catch. Most signals have specific meaning to *init*, usually requesting it to change its current state in some way. There is one signal, the 'Q' signal, which is used just to waken *init* and cause it to scan its database. This is often issued after a change has been made to the database so that *init* will put the new change into effect immediately. If this was not done, the change would not become effective until *init* had awakened for some other reason. Other than during the initialization phase, it is solely with signals that the system administrator controls the internal level at which *init* is running.

#### 2.4 Normal Operational Behavior

*Init* scans */etc/inittab* once or twice for each event which wakes it up. If it is in the *boot* or *powerfail* state, it scans the table once, looking for entries of these types, and then switches itself back to a *normal* state and scans again.

Its first action in the normal state is to scan */etc/inittab* and remove all processes which are currently active and should not be at the current level. *Init* employs one of two methods when killing its child processes depending on whether it is changing levels or not. If *init* is not changing levels, it forks a child process for each child that needs to be killed, and has that child process send the signals to the process targeted for extinction. Killing a process involves sending it two signals. First a SIGTERM signal (15), is sent so that it can clean up after itself and die gracefully. After waiting the amount of time defined as TWARN (the default value is 20 seconds), a SIGKILL signal (9), is sent, which guarantees that the child will die, if it hasn't done so already. Forking a child to do the killing has the advantage that the main *init* process need not wait for all the processes it is killing to die before beginning the spawning of new processes. The disadvantage is that if many processes were being killed this way, there would be a very real chance of the operating system process table filling up, which causes the *fork* system call to fail. This in turn would upset *init* at the very least and cause it to have to wait anyway. For this reason, when *init* is changing levels, it assumes that it may have many processes to terminate and so it sends the signals itself, waits for the required 20 seconds, and sends the final termination signals, before continuing. Once the old processes have been removed, *init* makes an entry in its accounting files if it is changing levels. At this point it either enters the single-user level or rescans its database looking for processes that need to be spawned at the current level and in the current state. In the normal state of operation *init* is looking for entries whose types are *off*, *once*, *wait*, or *respawn*.

With the completion of the scan of the database in the normal state, *init* is ready to wait for another event. To ensure that a user who just logged off has had his or her files updated to the disk and to insure that the bookkeeping is also updated to the disk, *init* performs a *sync* system call and then pauses until it is awakened again for some new reason.

If *init* finds that it is being requested to switch to the single-user level when it awakens from the pause, it saves all the *ioctl* information about the system console in the file *letchiocl.syscon* before proceeding to remove all its other children. It does this so that if the system is being taken down, the new *init* process will know how to set up the system console to talk to it. It is a convenient feature to not have to change the baud rate and terminal specifications if you are rebooting a system remotely. Because *init* preserves the *ioctl* state of the system console across system reboots, messages coming out during reboots are legible to the operator, no matter where the system console happens to be linked.



All written messages from *init* are sent to */dev/syscon*. In reality, *init* itself does not send the message, but forks a child to send the message. This is because *init* must never open a terminal line or it will be assigned a controlling terminal. Since *init* has no controlling terminal, it can spawn *getty* processes which initially have no controlling terminal. When such a *getty* opens its assigned terminal, the terminal becomes the controlling terminal for it and its children. In the one instance *init* needs input from the system administrator during the initialization phase. In this case, the child process which is asking for the run level opens */dev/systty*, which is always the physical system console, before opening */dev/syscon*, the virtual system console. This causes */dev/systty* to be the child's controlling terminal. Thus, should the computer be coming up, */dev/syscon* not be linked to */dev/systty*, and */dev/syscon* be down (perhaps because the datalink went down during the reboot), it is possible for a person at */dev/systty* to regain control by typing a <DEL> character. This causes a SIGINT signal (2) to be sent to the child process, which will relink */dev/systty* to */dev/syscon* and ask again for a run level, this time at the physical system console.

## 2.5 Setting Tunable Variables

*Init* has several tunable timing constants that can be adjusted when it is compiled.

**SLEEPTIME** *Init* guarantees that it will awaken occasionally even if the system is quite inactive. It does this by setting an alarm timer before going to sleep. The length of that timer is defined by SLEEPTIME, and is initially five minutes. Since *init* does a *sync* system call each time it wakes, this guarantees that there will be a *sync* at least once every SLEEPTIME seconds.

**TWARN** TWARN is the number of seconds between the SIGTERM signal and the SIGKILL signal, when *init* is removing processes. It should be set long enough so that all processes who want to, can die gracefully on receipt of the SIGTERM signal. It is initially 20 seconds.

**NPROC** This is the size of the internal process table *init* uses to keep track of its child processes. It currently defaults to 100, though it can be passed in during compilation with the -D option. I recommend you set it to the size of the system's process table.

**WARNFREQUENCY** To prevent *init* from flooding the system console with error messages when its own internal process table is full, *init* only generates an error message once each WARNFREQUENCY times that it is unable to find a slot. Proper sizing of the internal process table should prevent this condition from ever occurring.

*Init* cannot directly tell if there is something wrong when it tries to fork and exec a command. It assumes that there is something wrong if it has to respawn a particular entry too often. There are three related defines controlling this feature, SPAWN\_LIMIT, SPAWN\_INTERVAL, and INHIBIT.

**SPAWN\_LIMIT** SPAWN\_LIMIT is the number of times a process may respawn in a certain interval of time before further respawns are inhibited.

**SPAWN\_INTERVAL** SPAWN\_INTERVAL is the interval of time in seconds that SPAWN\_LIMIT number of respawns must occur to cause inhibition of an entry. If an entry should respawn too often, a message is generated on the system console indicating which line in */etc/inittab* is at fault.

**INHIBIT** INHIBIT is the number of seconds of inhibition that will be applied to a process which has respawned too often.

SPAWN\_LIMIT and SPAWN\_INTERVAL should be set so that it is possible for *init* to respawn a process fast enough to cause inhibition, but not so low that it is possible to have a legal death of a process happen so rapidly that it is inhibited. The current limits are ten respawns in two minutes. The real problem is that when something like *getty* disappears, *init* becomes active trying to respawn many processes and never gets to respawn a single process often enough to set off the alarm. The INHIBIT limit is five minutes. Once an entry is inhibited, it is possible to restart it sooner than



INHIBIT seconds later by sending *init* the 'Q' signal. The normal problem is a typo in */etc/inittab*, and the normal procedure is to correct the typo and then do a "telinit Q" to cause *init* to attempt the spawning entry again.

## 2.6 Debugging Features

*Init* has some debugging features built in. There are three conditional debug flags, which allow various flavors of debugging to be enabled.

**UDEBUG** This flag causes *init* to be compiled in a form that can be run as a normal user process instead of as process 1. This allows a person to use *sdb* on it in a normal fashion and to not disturb the rest of the system while debugging or modifications are made and tested. There are differences in this user version of *init*. It assumes that *utmp*, *wtmp*, *inittab*, *ioctl.syscon*, and *debug* are all in the local directory instead of */etc*. It also writes to */dev/sysconx* and */dev/systtyx*, instead of */dev/syscon* and */dev/systty*. It does not process all signals in the same fashion that the real *init* does. Signals SIGINT, SIGQUIT, SIGIOT, and SIGTERM, which correspond to the signals to change to levels 2, 3, 4, and ignore are left in their default modes, so that it is possible to terminate the user "init" from a terminal. Signals SIGUSR1 and SIGUSR2, which are normally ignored by the real *init* are set to cause an *abort* for capturing cores of the debug *init*. The UDEBUG flag automatically sets the DEBUG flag, meaning that the first level of debug will be generated by the *init* and written into the file *debug* in the current directory.

**DEBUG** This flag causes a version of *init* to be produced that can be run as the real *init*, but which generates diagnostic messages about process removal, level changes, and accounting and writes them in the file */etc/debug*.

**DEBUG1** DEBUG1 causes the diagnostic output generated by DEBUG1 to be increased substantially. Specifically it produces messages about each process being spawned from *inittab*.

## 3. Getty

*Getty* is responsible for making appropriate setting of terminal characteristics and baud rate so that a user can communicate with the UNIX system. The most important of those features is the choice of a baud rate so that input and output make sense. In the old version of *getty*, there was a hardwired table in *getty* which controlled the search for the correct speed. The starting point in the search is specified by the arguments passed to *getty*. If there was some reason to change the baud rate search, *getty* had to be modified itself, and recompiled. In the new *getty*, the search is controlled by an ascii file, */etc/gettydefs*, and changing or augmenting the search behavior only requires that the file be edited.

### 3.1 Usage

*Getty* is normally started from */etc/inittab* by *init*. *Getty* takes from one to six arguments:

*getty* [-h] [-t time] line [speed\_label] [term\_type] [line\_disc]

-h This switch tells *getty* that it should not drop the Data Terminal Ready signal before resetting the line. This switch currently only works in the CB-UNIX system environment. Normally *getty* ensures that DTR goes down so that connections to the Develcon dataswitch will be disconnected everytime. The EIA protocol requires that a dataset see DTR drop and be reasserted before answering another call. It is possible for *getty* to come back on a line before all the processes spun off by the previous user have died and closed their connections to the line. In this case, DTR would not drop if *getty* didn't insure it. This switch is required for programs like *ct*, which initiate a call from the computer to a user (instead of the user calling the computer), putting a *getty* on the resulting connected line. Without the -h switch, the *getty* would immediately disconnect the user again.



- t This switch specifies that the *getty* should die after the specified number of seconds if nothing is typed. This prevents datasets from being tied up if someone isn't actually logging in after they've gotten connected.
- line *Line* is the name of the terminal line, which *getty* is to open and set up. It is minus */dev/* since *getty* does a *chdir* to the */dev* directory and expects to find it in that directory.
- speed\_label The *speed\_label* is usually something like "1200" or "9600", which appears to directly specify a baud rate, but in reality can be anything since it really is a label of an entry in */etc/gettydefs* for which *getty* looks. It specifies the entry *getty* will start with when trying to find an appropriate speed to for the terminal. It defaults to "300" if there is none given.
- term\_type The *term\_type* specifies which terminal discipline is to be used. If this is specified, the virtual terminal protocol becomes immediately effective on the line. Typical types might be "vt100", "hp45", or "tek". Whatever type is specified, it must be a terminal handler that has been compiled into the operating system to be effective. This argument is given for lines that are hardwired to the computer.
- line\_disc The *line discipline* is the last thing that can be specified. The most common is "half" or "half\_duplex", when there is a half duplex terminal coming into the computer. This causes the appropriate line discipline to be associated with the line.

### 3.2 The Database: */etc/gettydefs*

Whenever *getty* is invoked it references its database to determine certain information about how to set up the line. Each entry in the database has a fixed format.

label# initial flags # final flags # login msg #nextlabel

*Getty* matches its *speed\_label* argument against the "label" field. It stops searching when it finds an entry with a label that matches. The entry specifies how the terminal is supposed to be setup during the initial phase, the phase when *getty* prints out the "login msg" and reads in the user's login name, and the final phase, when *getty* exec's the *login* program to continue to the *login* process. The baud rate is specified as an *ioctl* flag in both the initial and final flags fields.

The flags themselves are strings matching the define variables found in */usr/include/termio.h*. It should be noted that these flags may be partially or totally overridden if there is a terminal type specified. When a terminal type is enabled, it resets various flags to suitable conditions automatically.

During the initial phase, *getty* always puts the terminal into a non-echoing *raw* mode. This allows it to take each character as it comes in and infer certain things about the terminal. For instance, if it sees upper case alphabetic characters, but no lower case, it then assumes that the terminal is upper case only and sets it up in the final configuration so that the upper to lower case conversions are made. Also if the speed is wrong it will get a <NULL> character (or <ESC><NULL> character if a terminal type is set) if there is a framing or parity error. This means that the speed is wrong and another speed should be tried.

The typical "initial flags" would only include the speed, for example "B1200 CS7 PARENB HUPCL". "CS7 PARENB" sets the line for 7 bits, even parity characters. "HUPCL sets the line to hangup on close. Typical final flags would be "B1200 SANE IXANY TAB3". "SANE" is not a real flag found in the header file, but a collection of *ioctl* flags used for normal terminal behavior. "IXANY" permits the use of any character to restart output. "TAB3" says to expand tabs on output.

The "login msg" field is the message that *getty* will print before waiting for the user to enter his or her login name. It may contain anything desired and *getty* understands normal special character conventions so that "\n" means <lf> as does "\012". On systems that are not using the terminal



handlers and where lines are hardwired, people have been known to make up special entries for different terminal types, for example:

```
vt100-2400# B2400 # B2400 SANE TAB3 7953OGIN: #vt100-1200
# 33[H 33[2JAMACCS System B
```

where the "login msg" contains the special vt100 characters required to clear the screen. Notice also that the entry can take more than one line. Entries are delimited by a blank line. Lines that begin with a pound sign (#) are ignored so that comments may be added to the file.

The "next\_label" field tells *getty* which entry to try next if it gets an indication that the speed is wrong. In the above example it would look for an entry with the name "vt100-1200" if this one wasn't at the proper speed. Normally the entries don't contain terminal specific information, and the various speed choices are linked together in a closed circle of some sort. For example it is common to have 9600 -> 4800 -> 2400 -> 1200 -> 300 -> 9600. In this way, no matter where you enter the circle, sooner or later you should be able to get to the speed that is correct for your terminal.

To enable the system administrator to check the database for readability by *getty*, there is a checking mode in which *getty* can be run.

```
getty -c gettydefs_like_file
```

When *getty* is run in this mode, it scans the entire input file specified and deciphers each entry, printing out the resulting modes that it will set. If it finds a line that it cannot read, it prints an appropriate message, which allows the administrator to correct the entry. By this mechanism it is possible to avoid installing a misformatted *gettydefs* file and have it tie up the system.

Also as a safety measure, should *getty* be unable to find */etc/gettydefs*, it does have a one fallback entry built in. Should *gettydefs* disappear for some reason, a user could still log in at 300 baud, since this is the default setting in the built-in entry.

### 3.3 Operational Behavior

As has been shown earlier, *getty* sets up a line as specified by an entry from */etc/gettydefs* and from any additional arguments, outputs the "login msg" field, and then tries to read the user's login name from the line. During the input of the login name, *getty* checks for speed mismatches that the operating system will report as a <NULL> character. If such a mismatch occurs, *getty* tries the next speed specified by the current entry, and repeats the whole sequence. Also while reading in the login name, *getty* makes a guess whether the terminal is upper case only. If it sees some upper case characters, but no lower case characters, it assumes that the terminal is upper case only and sets the *ioctl* state of the line to translate upper case letters to lower case on input, and lower to upper case on output.

An addition has been made to *getty* and *login*, which allows for environmental variables to be set up at the time a user enters his or her login name. This allows users to control the behavior of their *.profile* at the time they specify their login names. *Getty* executes the *login* program by passing all the separate words given it in response to the login message as arguments to *login*. If for example, the user responded with "jls f", then *getty* would execute "login jls f" as its final action. See the *login* section to see how this modifies the commands behavior.

## 4. login

Unlike *init* and *getty*, *login* did not require a great deal of modification. The only required change was that it should write to */etc/utmp* and */etc/wtmp* in the new format. This change was minor. At the time this change was made, a change visible to the user was also made: the ability to add to the environment. This change was added as a convenience. It allows the user to modify the behavior of his or her *.profile* by having environmental variables set which the *.profile* script knows about.



The basic change was that any additional words provided in response to the basic "login:" query are placed in the environment of the *sh* executed by *login* as its last act in the following way. If the word does not contain an '=', a shell variable of the type "Ln=word" is created. "n" is a number starting at 0 and for each new environment variable it is incremented by one. If the word does contain an '=', then the whole string is passed in the environment unchanged. For example, "TERM=2621" would be placed in the environment unchanged and the shell variable \$TERM would be defined as "2621".

To preserve security, there are a couple of exceptions. It is not possible to change the shell variables \$PATH or \$SHELL by this mechanism. That means that a restricted shell will remain restricted and that the user cannot gain access to commands that might allow him to avoid the usual restrictions of *rsh*.

## 5. who

*Who*(1) is the program that reads the history files maintained by *init*, *getty*, and *login*. Since the format of these files was changed substantially, it was necessary to change *who*. In the process some additional features were added to *who* so that it would convey more useful information to users. The standard usage for *who* is:

**who [-uTlpdrbtas] [[am i] or [utmp\_like\_file]]**

- u This returns a listing of useful information for all the users. This information includes login time, activity, pid and comment from *inittab* file.
  - T Report the writability state of the terminal for that entry.
  - l Report all entries that are living *getty* processes.
  - p Report all entries for living children of *init* excluding *getty* and decendents of *getty*.
  - d Report all the entries for processes that have died.
  - b Report the boot time entries that *init* has made. In */etc/utmp* there is only one such entry.
  - r Report the run level entries that *init* has made. In */etc/utmp* there is only one such entry, the current run level entry. The current state, the number of times in that state, and the previous state are also reported.
  - t Report the change of date entries that have been made by the *date*(1) command when the clock was reset. These are required in the history file, */etc/wtmp*, if accounting is to be done.
  - a Report all the entries.
  - s Report information for all users in short form, this is the default.
- If no file is specified, then */etc/utmp* is assumed. The **who am i** sequence returns the entry for the user typing the command.

There are various output formats for the different kinds of entry. In particular, entries for users and *getty* processes list the amount of time since output to the terminal occurred. This is often of interest since it shows other users whether someone is actually working at a terminal or not. The comment field at the end of the entry from */etc/inittab* is also included, which can conveniently be set up to be the location of the terminal. Dead entries report the exit status for the process that died. This can be of use, since it shows whether the process terminated abnormally or not.

## 6. Other Affected Programs

All programs accessing the accounting files were affected by the new *utmp* structure. In particular, *date*(1) makes two entries indicating the old time and new time, whenever it changes the system clock. Also affected are the commands in */usr/lib/acct*, which produces reports based on the information in */etc/wtmp*.



## 7. utmp format

A major change in going to the new *init* was that it uses a different format in writing out its records in */etc/utmp* and */etc/wtmp*. The new format is:

```
/*      <sys/types.h> must be included.      */

#define UTMP_FILE      "/etc/utmp"
#define WTMP_FILE      "/etc/wtmp"
#define ut_name        ut_user

struct utmp
{
    char ut_user[8];      /* User login name */
    char ut_id[4];      /* /etc/lines id(usually line #) */
    char ut_line[12];    /* device name (console, lxxx) */
    short ut_pid;        /* process id */
    short ut_type;      /* type of entry */
    struct exit_status
    {
        short e_termination; /* Process termination status */
        short e_exit;        /* Process exit status */
    }
    ut_exit;            /* The exit status of a process
                        * marked as DEAD_PROCESS.
                        */
    time_t ut_time;     /* time entry was made */
};

/*      Definitions for ut_type      */

#define EMPTY          0
#define RUN_LVL        1
#define BOOT_TIME      2
#define OLD_TIME        3
#define NEW_TIME        4
#define INIT_PROCESS    5 /* Process spawned by "init" */
#define LOGIN_PROCESS   6 /* A "getty" process waiting for login */
#define USER_PROCESS    7 /* A user process */
#define DEAD_PROCESS    8
#define ACCOUNTING      9

#define UTMAXTYPE      ACCOUNTING /* Largest legal value of ut_type */

/*      Special strings or formats used in the "ut_line" field when
/*      accounting for something other than a process.
/*      No string for the ut_line field can be more than 11 chars +
/*      a NULL in length.

#define RUNLVL_MSG      "run-level %c"
#define BOOT_MSG        "system boot"
#define OTIME_MSG       "old time"
#define NTIME_MSG       "new time"
```

The *ut\_type* field completely identifies the type of entry, the *ut\_id* field only contains the "id" as found in the "id" field of */etc/inittab*. The *ut\_line* field was expanded and freed so that it can



contain things like *console* or other things that are not of the form */dev/lxxx*. Finally *ut\_exit* contains the exit status of processes that *init* has spawned and that have subsequently died.



THE UNIVERSITY OF CHICAGO PRESS  
54 EAST LAKE STREET, CHICAGO, ILL. 60601-3043  
TEL: (312) 937-1221 FAX: (312) 937-1441



# UUCP Tutorial

-

Documentation-No.:



UNIVERSITÄT

**Trademarks:**

MUNIX, CADMUS  
DEC, PDP  
UNIX

for PCS  
for DEC  
for Bell Laboratories

Copyright 1984 by  
PCS GmbH, Pfälzer-Wald-Strasse 38, D-8000 München 90, tel. (089) 67804-0

The information contained herein is the property of PCS and shall neither be reproduced in whole or in part without PCS's prior written approval nor be implied to grant any license to make, use or sell equipment manufactured herewith.

PCS reserves the right to make changes without notice in the specifications and materials contained herein and shall not be responsible for any damages (including consequential) caused by reliance on the materials presented.



## CONTENTS

|                                               |   |
|-----------------------------------------------|---|
| 1. Introduction . . . . .                     | 1 |
| 2. The Uucp Network . . . . .                 | 1 |
| 2.1 Network Hardware . . . . .                | 1 |
| 2.2 Network Topology . . . . .                | 2 |
| 2.3 Forwarding . . . . .                      | 2 |
| 2.4 Security . . . . .                        | 3 |
| 2.5 Software Structure . . . . .              | 3 |
| 2.6 Rules of the Road . . . . .               | 3 |
| 2.7 Special Places: The Public Area . . . . . | 4 |
| 2.8 Permissions . . . . .                     | 4 |
| 3. Network Usage . . . . .                    | 4 |
| 3.1 Name Space . . . . .                      | 4 |
| 3.2 Forwarding Syntax . . . . .               | 5 |
| 3.3 Types of Transfers . . . . .              | 6 |
| 3.4 Remote Executions . . . . .               | 6 |
| 3.5 Spooling . . . . .                        | 7 |
| 3.6 Notification . . . . .                    | 7 |
| 3.7 Tracking and Status . . . . .             | 7 |
| 3.8 Job Status . . . . .                      | 8 |
| 3.9 Network Status . . . . .                  | 8 |
| 3.10 Job Control . . . . .                    | 8 |
| 4. Utilities That Use Uucp . . . . .          | 9 |
| 4.1 Mail . . . . .                            | 9 |
| 4.2 Netnews . . . . .                         | 9 |
| 4.3 Other Applications . . . . .              | 9 |
| 5. Conclusion . . . . .                       | 9 |



1. The first part of the report deals with the general situation of the country and the progress of the work during the year. It is divided into two main sections: the first section deals with the general situation of the country and the progress of the work during the year, and the second section deals with the specific results of the work.

2. The second part of the report deals with the specific results of the work. It is divided into three main sections: the first section deals with the results of the work in the field of agriculture, the second section deals with the results of the work in the field of industry, and the third section deals with the results of the work in the field of commerce.

3. The third part of the report deals with the conclusions and recommendations. It is divided into two main sections: the first section deals with the conclusions and the second section deals with the recommendations.

4. The fourth part of the report deals with the appendix. It contains the following items:

- a. A list of the names of the members of the committee.
- b. A list of the names of the members of the sub-committee.
- c. A list of the names of the members of the working group.
- d. A list of the names of the members of the advisory committee.
- e. A list of the names of the members of the executive committee.
- f. A list of the names of the members of the secretariat.
- g. A list of the names of the members of the finance committee.
- h. A list of the names of the members of the legal committee.
- i. A list of the names of the members of the technical committee.
- j. A list of the names of the members of the social committee.
- k. A list of the names of the members of the cultural committee.
- l. A list of the names of the members of the sports committee.
- m. A list of the names of the members of the health committee.
- n. A list of the names of the members of the education committee.
- o. A list of the names of the members of the science committee.
- p. A list of the names of the members of the art committee.
- q. A list of the names of the members of the literature committee.
- r. A list of the names of the members of the music committee.
- s. A list of the names of the members of the drama committee.
- t. A list of the names of the members of the cinema committee.
- u. A list of the names of the members of the radio committee.
- v. A list of the names of the members of the television committee.
- w. A list of the names of the members of the press committee.
- x. A list of the names of the members of the publishing committee.
- y. A list of the names of the members of the book committee.
- z. A list of the names of the members of the library committee.



# Uucp Tutorial

## 1. Introduction

The *uucp* network has provided a means of information exchange between UNIX<sup>1</sup> systems over the DDD network for several years [1]. This document is an attempt to provide the novice user with the background to make use of the network.

While it is perfectly reasonable for a user to assume that knowledge of the mechanism for transmission of information between UNIX Systems is unnecessary, in practice, it is often useful to understand basic principles so that the best possible use of the network can be made. Two earlier documents describe the goals and the details of an early implementation of *uucp* [1, 2]. This document covers much of the same ground but reflects the improvements that have been made to the system and the information is tailored to the novice user.

The first half of the document discusses concepts. The second half explains the use of the user level interface to the network and provides numerous examples.

## 2. The Uucp Network

The *uucp*(1) network is a network of UNIX systems that allows file transfer and remote execution to occur on a network of UNIX Systems. The *extent* of the network is a function of both the interconnection hardware and the controlling network software. Membership in the network is tightly controlled via the software to preserve the integrity of all members of the network. The following sections describe the topology, services, operating rules, etc. of the network in detail to provide a framework for discussing use of the network.

### 2.1 Network Hardware

Uucp(1) was originally designed as a dialup network so that systems in the network could use the DDD network to communicate with each other. The three most common methods of connecting systems are,

1. Two UNIX Systems can be directly connected by cross-coupling (via a null modem) two of the computers ports. This means of connection is useful for only short distances (several hundred feet can be achieved although the RS232 standard specifies a much shorter distance) and is usually run at high speed (9600 baud). These connections run on asynchronous terminal ports.
2. A modem (using a private line or a limited distance modem) can be used to *directly* connect processors over a private line (using 103 or 212 type datasets).
3. Lastly, and more commonly, a processor can use a modem and an automatic calling unit (ACU) to use the DDD network to contact another system. This is by far the most common interconnection method and makes available the largest number of connections.

In principle *uucp* could be extended to use higher speed media (e.g., HYPERchannel<sup>2</sup>, Ethernet<sup>3</sup>, etc.) and this possibility is being explored for future UNIX releases. Some sites already support local modifications to *uucp* for using Datakit, X.25 (permanent virtual circuits) and calling through dataswitches.

---

1. UNIX is a Trademark of Bell Telephone Laboratories, Incorporated.

2. HYPERchannel is a Trademark of Network Systems Corporation.

3. Ethernet is a Trademark of Xerox Corp.



## 2.2 Network Topology

A large number of connections between systems are possible via the DDD network. The topology of the network is, however, determined by both the hardware connections and the software that controls the network. The next two sections deal with how that topology is controlled.

**2.2.1 Hardware Topology** As discussed earlier, it is possible to build a network using permanent or dial up connections. In Figure 1, a group of systems (A, B, C, D and E) are shown as connected via hardwired lines (all systems are assumed to have some answer only datasets so remote users or systems can be connected). A few systems have automatic calling units (K, D, F and G) and one system (H) does not have any capability for calling other systems. Users should be aware that the network consists of a series of point to point connections (A-B, B-C, D-B, E-B) even though it appears in Figure 1 that A and C are directly connected through B. The following observations can be made,

1. System H is isolated. It can be made part of the network by arranging for other systems to *poll* it at fixed intervals. This is an important concept to remember since transfers from systems that are *polled* will not leave the system until that system is called by a polling system.
2. Systems K, F, G and D can easily reach all other systems since they have calling units.
3. If system A (E or G) wishes to send a file to H (K, F or G) it must first send it to D (via system B) since D is the only system with a calling unit.

**2.2.2 Software Topology** The hardware capability of systems in the network defines the *maximum* number of connections in the network. The software at each node restricts the access by other systems and thereby defines the extent of the network. The systems of Figure 1 can be configured so that conceptually they appear as a network of systems that have equal access to each other or some restrictions can be applied. As part of the security mechanism used by *uucp*(1), the extent of access that other systems have can be controlled at each node. Figure 2a,b shows how the network might appear at one node. Access is available from all systems in Figure 2a, however, in Figure 2b some of the systems have been configured to have greater or less access privileges than others (i.e., system C, E, G have one set of access privileges, systems F and B have another set, etc.).

*Uucp* uses the UNIX password mechanism coupled with a system file (*/usr/lib/uucp/L.sys*) and a filesystem permission file (*/usr/lib/uucp/USERFILE*) to control access between systems. The password file entries for *uucp* (usually, *luucp*, *nuucp*, *uucp*, etc.) allow only those remote systems that know the passwords for these id's to access the local system. (Great care should be taken in revealing the password for these *uucp* logins since knowing the password allows a system to join the network.) The system file (*/usr/lib/uucp/L.sys*) defines the remote systems that a local host knows about. This file contains all information needed for a local host to contact a remote system (including system name, password, login sequence, etc.) and as such is protected from viewing by ordinary users.

In summary, while the available hardware on a network of systems determines the connectivity of the systems, the combination of *password* file entries and the *uucp* system files determines the extent of the network.

## 2.3 Forwarding

One of the recent additions to *uucp* (for UNIX System V) is a limited forwarding capability whereby systems that are part of the network can forward files through intermediate nodes. For example, in Figure 1 it is possible to send a file between node A and C through intermediate node B. For security reasons, when forwarding, files may only be transmitted to the *public* area (or fetched from the remote systems *public* area).



## 2.4 Security

The most critical feature of any network is the security that it provides. Users are familiar with the security that UNIX provides in protecting files from access by other users and in accessing the system via *passwords*. In building a network of processors, the notion of security is widened because access by a wider community of users is granted. Access is granted on a *system* basis (that is, access is granted to *all* users on a remote system). This follows from the fact that the process of sending (receiving) a file to (from) another system is done via daemons that use one special user id(s). This user id(s) is granted (denied) access to the system via the *uucp* system file (*/usr/lib/uucp/L.sys*) and the areas that the system has access to is controlled by another file (*/usr/lib/uucp/USERFILE*). For example, access can be granted to the entire file system tree or limited to specific areas.

## 2.5 Software Structure

The *uucp(1)* network is a batch network. That is, when a request is made, it is spooled for later transmission by a daemon. This is important to users because the success or failure of a command will only be known at some later time via *mail* notification. For most transfers, there is little trouble in transmitting files between systems, however, transmissions are occasionally delayed or fail because a remote system can't be reached.

## 2.6 Rules of the Road

There are several rules by which the network runs. These rules are necessary to provide the smooth flow of data between systems and to prevent duplicate transmissions and lost jobs. The following sections outline what these rules are and what their influence on the network is.

**2.6.1 Queuing** Jobs submitted to the network are assigned a sequence number for transmission. Jobs are represented by a file (or files) in a common spool directory (*/usr/spool/uucp*). When a file transfer daemon (*uucico*) is started to transmit a job, it selects a system to contact and then transmits *all* jobs to that system. Before breaking off the conversation, any jobs to be received from that remote system are accepted. The system selected as the one to contact is randomly selected if there is work for more than one system (in releases of *uucp(1)* prior to UNIX System V, the first system appearing in the spool directory was selected so preference was given to the most recently spawned jobs). *Uucp* may be sending to or receiving from many systems simultaneously. The number of incoming requests is only limited by the number of connections on the system and the number of outgoing transfers is limited by the number of ACUs (or direct connections).

**2.6.2 Dialing and the DDD Network** In order to transfer data between processors that are not directly connected, an auto dialer is used to contact the remote system. There are several factors that can make contacting a remote system difficult.

1. All lines to the remote system may be busy. There is a mechanism within *uucp* that restricts contact with a remote to certain times of the day (week) to minimize this problem.
2. The remote system may be down.
3. There may be difficulty in dialing the number (especially if a large sequence of numbers involving access through PBX's is involved). The dialing algorithm tries dialing a number twice and the algorithm used to dial remote systems is not perfect, particularly when intermediate dial tones are involved.

**2.6.3 Scheduling and Polling** When a job is submitted to the network an attempt to contact that system is made *immediately*. Only one conversation can exist between the same two systems at a time.

Systems that are *polled* can do nothing to force immediate transmission of data. Jobs will only be transmitted when the system is polled (hourly, daily, etc.) by a remote system.



**2.6.4 Retransmissions and Hysteresis** The *uucp* network is fairly tenacious in its attempt to contact remote systems to complete a transmission. To prevent *uucp* from continually calling systems that are unavailable, *hysteresis* is built into the algorithm used to contact other systems. This mechanism forces a minimum fixed delay (specifiable on a per system basis) to occur before another transmission can take place to that system.

**2.6.5 Purging and Cleanup** Users should be aware that transfers that cannot be completed after a defined period of time (72 hours is the value that is set when the system is distributed) are deleted and the user is notified.

## 2.7 Special Places: The Public Area

In order to allow the transfer of files to a system that a user may not have a login on, the *public* directory (usually kept in */usr/spool/uucppublic*) is available as an area with general access privileges. When receiving files in the *public* area, the user should dispose of them quickly as the administrative portion of *uucp* purges this area on a regular basis.

## 2.8 Permissions

**2.8.1 File Level Protection** In transferring files between systems, users should make sure that the destination area is writeable by *uucp*. The *uucp* daemons will preserve execute permission between systems and assign permission 0666 to transferred files.

**2.8.2 System Level Protection** The system administrator at each site determines the global access permissions for that processor. Thus, access between systems may be confined to only some sections of the filesystem by an administrator.

**2.8.3 Forwarding Permissions** The forwarding feature is a new addition to the *uucp* package. Users of this feature should understand that,

1. If forwarding is attempted through a node that is running an old version of *uucp(1)*, the transmission will not succeed.
2. Nodes that allow forwarding can restrict the forwarding feature in several ways.
  - a. Allow forwarding for only certain users.
  - b. Prevent forwarding to certain destination nodes (e.g., *nodeblia*).
  - c. Allow forwarding for selected source nodes.
3. The most important restriction is that forwarding is allowed only for files sent to or fetched from the *public* area.

## 3. Network Usage

The following sections discuss the user interface to the network and give examples of command usage.

### 3.1 Name Space

In order to reference files on remote systems, a syntax is necessary to uniquely identify a file. The notation must also have several defaults to allow the reference to be compact. Some restrictions must also be placed on pathnames to prevent security violations. For example, pathnames may not include "." as a component because it is difficult to determine whether the reference is to a restricted area.

**3.1.1 Naming Conventions** *Uucp* uses a special syntax to build references to files on remote systems. The basic syntax is,

*system-name/path-name*

where the *system-name* is a system that *uucp(1)* knows about. The *path-name* part of the name



may contain any of the following,

1. A fully qualified *path-name* such as

*xnode!husrlyoufile*

The *path-name* may also be a directory name as in

*xnode!husrlyouldirectory*

2. The login directory on a remote may be specified by use of the `~` character. The combination `~user` references the login directory of a user on the remote. For example,

*xnode!~admfile*

would expand to

*xnode!husr/sys/admfile*

if the login directory for user *adm* on the remote system is *husr/sys/adm*.

3. The *public* area is referenced by a similar use of the prefix `~huser` preceding the pathname. For example,

*xnode!~lyoufile*

would expand to

*xnode!husr/spool/hucphyoufile*

if *husr/spool/hucp* is used as the spool directory.

4. Pathnames not using any of the combinations or prefixes discussed above are prefixed with the current directory (or the login directory on the remote). For example,

*xnode!file*

would expand to

*xnode!husrlyoufile*

The naming convention can be used in reference to either the *source* or *destination* file names.

### 3.2 Forwarding Syntax

The newest feature of *uucp* is the ability of *uucp* to allow files to be passed between systems via intermediate nodes. This is done via a variation of the *bang (!)* syntax that describes the path to be taken to reach that file. For example, a user on system *a* wishing to transmit a file to system *e* might specify the transfer as,

*uucp file b!c!d!e!~lyoufile*

if the user desires the request to be sent through *b*, *c* and *d* before reaching *e*. Note that the pathname is the path that the file would take to reach node *e*. Note also that the destination *must* be specified as the *public* area. Fetching a file from another system via intermediate nodes is done similarly. For example,



```
uucp b!c!d!e!~!you!file x
```

fetches *file* from system *e* and renames it *x* on the local system. The forwarding prefix is the path from the local system not the path from the remote to the local system. The forwarding feature may also be used in conjunction with remote execution. For example,

```
uux xnode!uucp ynode!pnode!husr/spool/uucppublic!file x
```

sends a request to *xnode* to execute the *uucp* command to copy a file from *pnode* to *x* on *xnode*.

### 3.3 Types of Transfers

*Uucp(1)* has a very flexible command syntax for file transmission. The following sections give examples of different combinations of transfers.

**3.3.1 Transmissions of Files to a Remote** Any number of files can be transferred to a remote system via *uucp(1)*. The syntax supports the *\**, *?* and *[.]* meta characters. For example,

```
uucp *.!ch! xnode!dir
```

transfers *all* files whose name ends in *c* or *h* to the directory *dir* in the users login directory on *xnode*.

**3.3.2 Fetching Files From a Remote** Files can be fetched from a remote system in a similar manner. For example,

```
uucp xnode!*.!ch! dir
```

will fetch all files ending in *c* or *h* from the users login directory on *xnode* and place the copies in the subdirectory *dir* on the local system.

**3.3.3 Switching** Transmission of files can be arranged in such a way that the local system effectively acts as a switch. For example,

```
uucp ynode!files xnode!filed
```

will fetch *files* from the users login directory on *ynode*, rename it as *filed* and place it in the login directory on *xnode*.

**3.3.4 Broadcasting** Broadcast capability (that is, copying a file to many systems) is *not* supported by *uucp*, however, it can be simulated via a shell script as in

```
for i in xnode ynode mhtsd
do
    uucp file $i!broad
done
```

Unfortunately, one *uucp* command is spawned for each transmission so that it is not possible to track the transfer as a single unit.

### 3.4 Remote Executions

The remote execution facility allows commands to be executed remotely. For example,

```
uux "!diff xnode!/etc/passwd mhtsd!/etc/passwd > !pass.diff"
```

will *diff(1)* the password file on *xnode* and *mhtsd* and place the result in *pass.diff*.



### 3.5 Spooling

If a user wishes to continue modifying a file while a copy is being transmitted across the network, the *-c* option should be used. This forces a *copy* of the file to be queued. The default for *uucp* is not to queue copies of the files since it is wasteful of both cpu time and storage. For example, the following command will force the file *work* to be copied into the spool directory before it is transmitted.

```
uucp -c work xnode!~/you/work
```

### 3.6 Notification

The success or failure of a transmission is reported to users asynchronously via the *mail(1)* command. A new feature of *uucp* is to provide notification to the user in a file (of the users choice). The choices for notification are,

1. Notification returned to the requestors system (via the *-m* option). This is useful when the requesting user is distributing files to other machines and logging onto the remote machine to read mail is inconvenient.
2. A variation of the *-m* option is to force notification in a file (using the *-mfile* option where file is a file name). For example,

```
uucp -mans letc/passwd ynode!dev/null
```

will send the file *letc/passwd* to system *ynode* and place the file in the bit bucket (*dev/null*). The status of the transfer will be reported in the file *ans* as,

```
uucp job 0306 (8/20-23:08:09) (0:31:23) letc/passwd copy succeeded
```

3. *Uux(1)* always reports the exit status of the remote execution unless notification is suppressed (via the *-n* option). Notification can be sent to a different user on the remote (via the *-nuser* option).

### 3.7 Tracking and Status

The most pervasive change to the *uucp* package (for UNIX System V) has been revising the internal formatting of jobs so that each invocation of *uucp(1)* or *uux(1)* corresponds to a single job. It is now possible to associate a single job number with each command execution so that the job can be terminated or its status obtained.

**3.7.1 The Job ID** The default for the *uucp(1)* and *uux(1)* command is *not* to print the job number for each job. This was done for compatibility with previous versions of *uucp(1)* and to prevent the many shell scripts built around *uucp(1)* from printing job numbers. If the following environment variable

```
JOBNO=ON
```

is made part of the users environment and exported, *uucp(1)* and *uux(1)* will print the job number. Similarly, if the user wishes to turn job numbers off, the environment variable can be set as follows,

```
JOBNO=OFF
```

If the user wishes to force printing of job numbers without using the environment mechanism, the *-j* option can be used. For example,

```
uucp -j /etc/passwd ynode!dev/null  
uucp job 282
```



forces the job number (282) to be printed. If the `-j` option is not used, the ids of the jobs belonging to the user can be found by using the `uustat(lc)` command to obtain the job numbers. For example,

```
uustat
0282 tom ynode 08/20-21:47 08/20-21:47 JOB IS QUEUED
0272 tom ynode 08/20-21:46 08/20-21:46 JOB IS QUEUED
```

shows that the user has two jobs (282 and 272) queued.

### 3.8 Job Status

The `uustat(l)` command allows a user to check on one or all jobs that have been queued. The id printed when a job is queued is used as a key to query status of the particular job. An example of a request for the status of a given job is,

```
uustat -j0711

0711 tom ynode 07/30-02:18 07/30-02:18 JOB IS QUEUED
```

There are several status messages that may be printed for a given job; the most frequent ones are JOB IS QUEUED and JOB COMPLETED that have the obvious meanings. The manual page for `uustat(l)` lists the other status messages.

### 3.9 Network Status

The status of the last transfer to each system on the network can be found by using the `uustat(lc)` command. For example,

```
uustat -mall
```

will report the status of the last transfer to all of the systems known to the local system.

The output might appear as,

```
znode 08/10-12:35 CONVERSATION SUCCEEDED
mnode 08/20-17:01 CONVERSATION SUCCEEDED
anode 07/22-16:31 DIAL FAILED
nodeb 08/20-18:36 WRONG TIME TO CALL
knode 08/20-20:37 LOGIN FAILED
```

where the status indicates the time and state of the last transfer to each system. When sending files to a system that has not been contacted recently, it is a good idea to use `uustat(lc)` to see *when* the last access occurred as the remote system may be down or out of service.

### 3.10 Job Control

With the unique job id generated for each `uucp(l)` or `uux(l)` command, it is possible to control jobs in the following ways.

**3.10.1 Job Termination** A job that may consist of transferring many files from several different systems can be terminated using the `-k` option of `uustat(l)`. If any part of the job has left the system then only the *remaining* parts of the job on the local system will be terminated.

**3.10.2 Requeuing a Job** The `uucp(l)` package clears out its working area of jobs on a regular basis (usually every 72 hours) to prevent the buildup of jobs that cannot be delivered. To force the date of a job to be changed to the current date, thereby lengthening the time that `uucp` will attempt to transmit the job, the `-r` option can be used. It should be noted that the `-r` option does not impart



*immortality* to a job. Rather, it only postpones deleting the job during housekeeping functions until the next cleanup.

**3.10.3 Network Names** Users may find the names of the systems on the network via the *uname(1)* command. Only the *names* of the systems in the network are printed.

#### 4. Utilities That Use Uucp

There are several utilities that rely on *uucp(1)* or *uux(1)* to transfer files to other systems. The following sections outline the more important of these functions to increase awareness of the extent of the use of the network.

##### 4.1 Mail

The *mail(1)* command uses *uux(1)* to forward mail to other systems. For example, when a user types

```
mail xnode!tom
```

the *mail(1)* command invokes *uux(1)* to execute *rmail* on the remote system (*rmail* is a link to the *mail(1)* command). Forwarding mail through several systems (e.g., *mail a!b!tom*) does not use the *uucp(1)* forwarding feature but is simulated by the *mail(1)* command itself.

##### 4.2 Netnews

The *netnews(1)* command that is locally supported on many systems uses *uux(1)* in much the same way that *mail(1)* does to broadcast *network mail* to systems subscribing to news categories.

##### 4.3 Other Applications

The Office Automation System (OAS) uses *uux(1)* to transmit electronic mail between systems in a manner similar to the standard *mail(1)* command. Some sites have replaced utilities such as *lpr(1)*, *opr(1)*, etc. with shell scripts that invoke *uux(1)* or *uucp(1)*. Other sites use the *uucp* network as a backup for higher speed networks (e.g., PCL, NSC HYPERchannel, etc.).

#### 5. Conclusion

This document has outlined the rules that run the network and attempted to illustrate the use and power of *uucp*.



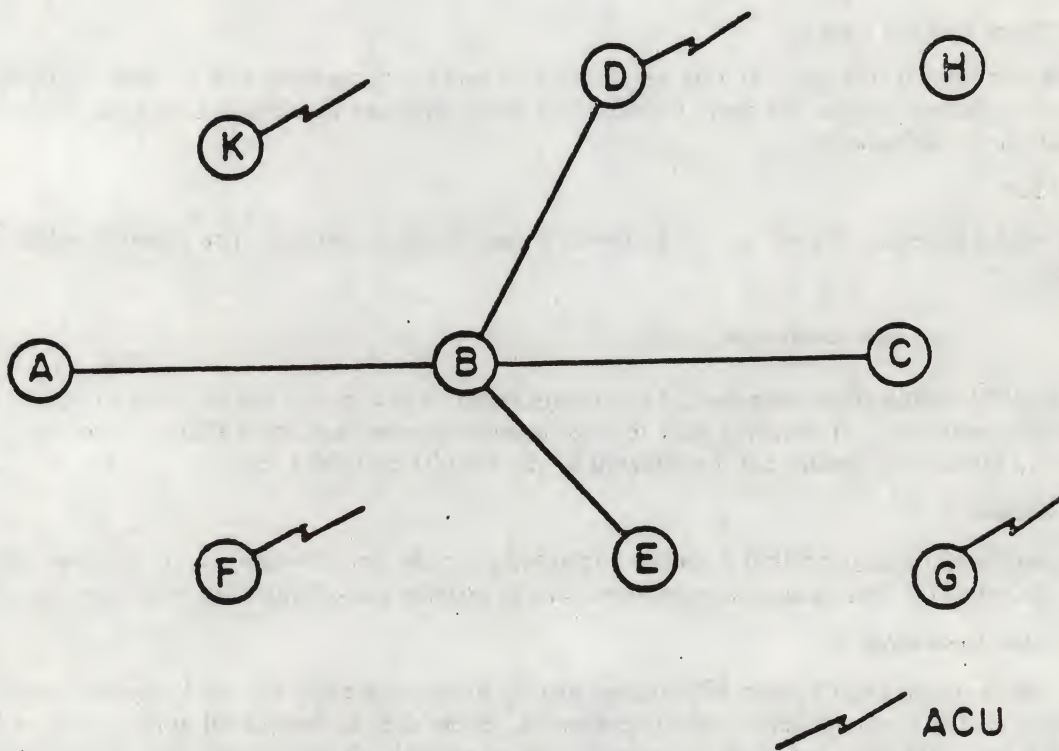


Figure 1 Uucp nodes.



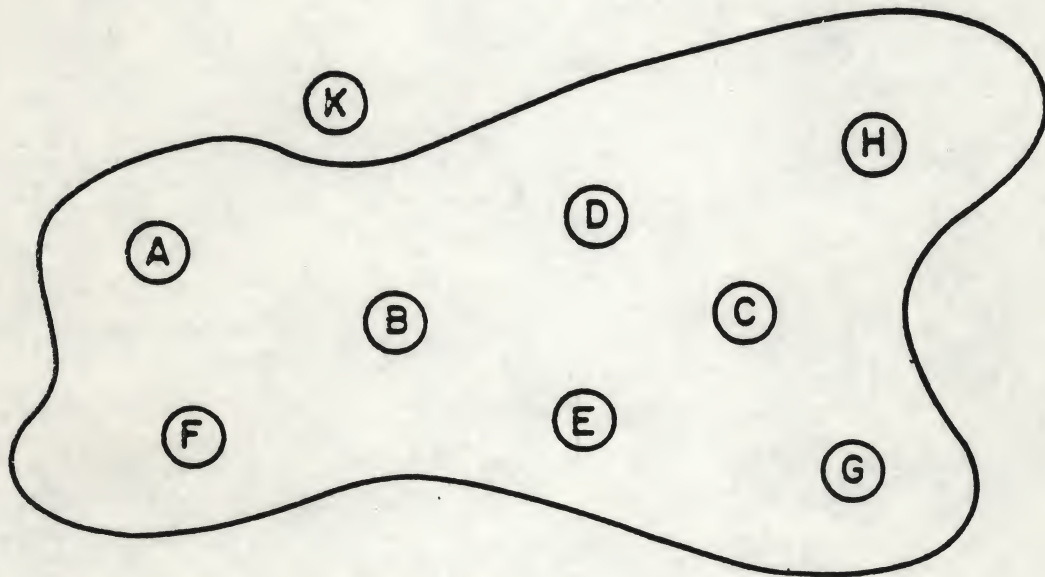


Figure 2a Uucp network excluding one node.

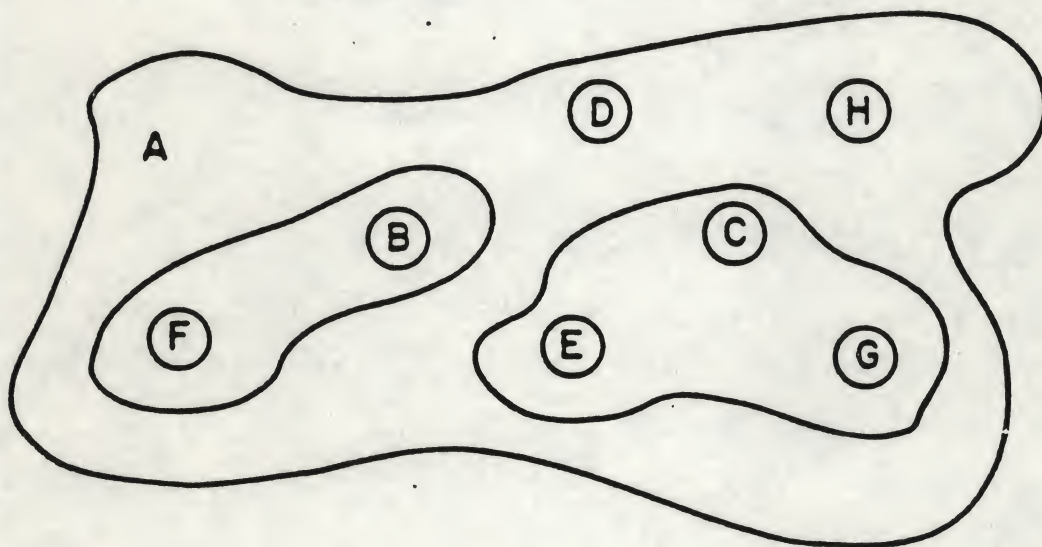


Figure 2b Uucp network with several levels of permissions.





Figure 1. A single irregular shape with 10 points.



Figure 2. A complex shape with two sub-shapes and 12 points.



# UUCP

## Administrator's Manual



Trademarks:

MUNIX, CADMUS  
DEC, PDP  
UNIX

for PCS  
for DEC  
for Bell Laboratories

Copyright 1984 by  
PCS GmbH, Pfälzer-Wald-Strasse 38, D-8000 München 90, tel. (089) 67804-0

The information contained herein is the property of PCS and shall neither be reproduced in whole or in part without PCS's prior written approval nor be implied to grant any license to make, use or sell equipment manufactured herewith.

PCS reserves the right to make changes without notice in the specifications and materials contained herein and shall not be responsible for any damages (including consequential) caused by reliance on the materials presented.



## CONTENTS

|                                                |   |
|------------------------------------------------|---|
| 1. Introduction . . . . .                      | 1 |
| 2. Planning . . . . .                          | 1 |
| 2.1 Extent of the Network . . . . .            | 1 |
| 2.2 Hardware and Line Speeds . . . . .         | 1 |
| 2.3 Maintenance and Administration . . . . .   | 2 |
| 3. A Quick Tour of the Uucp Software . . . . . | 2 |
| 4. Installation . . . . .                      | 2 |
| 4.1 Object Modules . . . . .                   | 2 |
| 4.2 Password File . . . . .                    | 2 |
| 4.3 Lines File . . . . .                       | 3 |
| 4.4 System File . . . . .                      | 3 |
| 4.5 Dialing Prefixes . . . . .                 | 5 |
| 4.6 USERFILE . . . . .                         | 5 |
| 4.7 Forwarding File . . . . .                  | 6 |
| 5. Administration . . . . .                    | 7 |
| 5.1 Cleanup . . . . .                          | 7 |
| 5.2 Polling Other Systems . . . . .            | 7 |
| 5.3 Problems . . . . .                         | 7 |
| 6. Debugging . . . . .                         | 8 |
| 7. Conclusion . . . . .                        | 8 |



# Introduction

1.1.1.1

1.1.1.2

1.1.1.3

1.1.1.4

1.1.1.5

1.1.1.6

1.1.1.7

1.1.1.8

1.1.1.9

1.1.1.10

1.1.1.11

1.1.1.12

1.1.1.13

1.1.1.14

1.1.1.15

1.1.1.16

1.1.1.17

1.1.1.18

1.1.1.19

1.1.1.20

1.1.1.21

1.1.1.22

1.1.1.23

1.1.1.24

1.1.1.25

1.1.1.26

1.1.1.27

1.1.1.28

1.1.1.29

1.1.1.30

1.1.1.31

1.1.1.32

1.1.1.33

1.1.1.34

1.1.1.35

1.1.1.36

1.1.1.37

1.1.1.38

1.1.1.39

1.1.1.40

1.1.1.41

1.1.1.42

1.1.1.43

1.1.1.44

1.1.1.45

1.1.1.46

1.1.1.47

1.1.1.48

1.1.1.49

1.1.1.50

1.1.1.51

1.1.1.52

1.1.1.53

1.1.1.54

1.1.1.55

1.1.1.56

1.1.1.57

1.1.1.58

1.1.1.59

1.1.1.60

1.1.1.61

1.1.1.62

1.1.1.63

1.1.1.64

1.1.1.65

1.1.1.66

1.1.1.67

1.1.1.68

1.1.1.69

1.1.1.70

1.1.1.71

1.1.1.72

1.1.1.73

1.1.1.74

1.1.1.75

1.1.1.76

1.1.1.77

1.1.1.78

1.1.1.79

1.1.1.80

1.1.1.81

1.1.1.82

1.1.1.83

1.1.1.84

1.1.1.85

1.1.1.86

1.1.1.87

1.1.1.88

1.1.1.89

1.1.1.90

1.1.1.91

1.1.1.92

1.1.1.93

1.1.1.94

1.1.1.95

1.1.1.96

1.1.1.97

1.1.1.98

1.1.1.99

1.1.1.100



## Uucp Administrator's Manual

### 1. Introduction

*Uucp* has been the mainstay of UNIX<sup>1</sup> system to UNIX system communication for several years [1, 2]. This document illustrates how a network is set up, the format of control files and administrative procedures. Administrators should be familiar with the *Uucp Users Tutorial* and the manual pages for each of the *uucp* related commands before reading this document.

### 2. Planning

In setting up a network of UNIX systems there are several considerations that should be taken into account *before* configuring each system on the network. The following sections attempt to outline the most important considerations.

#### 2.1 Extent of the Network

Some basic decisions about *access* to processors in the network must be made before attempting to set up the configuration files. If an administrator has control over only one processor and an existing network is being joined, then the administrator must decide what level of access should be granted to other systems. The other members of the network must make a similar decision for the new system. The UNIX system *password* mechanism is used to grant access to other systems. The file (*/usr/lib/uucp/USERFILE*) restricts access by other systems to parts of the filesystem tree and the file */usr/lib/uucp/L.sys* on the local processor determines how many other systems on the network can be reached.

When setting up more than one processor is involved, the administrator has control of a larger fraction of the network and can make more decisions about the setup of the network. For example, the network can be set up as a *private* network where only those machines under the direct control of the administrator can access each other. Granting *no* access to machines outside the network can be done if security is paramount, however, this is usually impractical. Very limited access can be granted to outside machines by each of the systems on the *private* network. Alternatively, access to/from the outside world can be confined to only one processor. This is frequently done to minimize the effort in keeping access information (passwords, phone numbers, login sequences, etc.) updated and to minimize the number of security holes for the private network.

#### 2.2 Hardware and Line Speeds

There are only two supported means of interconnection by *uucp* (1).

1. Direct connection using a null modem.
2. Connection over the DDD network.

Direct connection over private lines using X.25 is not fully supported in UNIX System V, although with the addition of one program *x25.login* it can be made operational. In choosing hardware, the equipment used by other processors on the network must be considered. For example, if some systems on the network have only 103 type (300 baud) datasets, then communication with them is not possible unless the local system has a 300 baud dataset connected to a calling unit. (Most datasets available on systems are 1200 baud.) If hardwired connections are to be used between systems, then the *distance* between systems must be considered since a null modem cannot be used when the systems are separated by more than several hundred feet (the limit for communication at

---

\* UNIX is a Trademark of Bell Telephone Laboratories, Incorporated.



9600 baud is about 800 to 1000 feet although the RS232 specification allows for less than fifty feet). Limited distance modems must be used beyond these distances or if noise on the lines becomes a problem.

### 2.3 Maintenance and Administration

There is a minimum amount of maintenance that must be provided on each system to keep the access files updated, to insure that the network is running properly and to track down line problems. When more than one system is involved, the job becomes more difficult because there are more files to update and because users are much less patient when failures occur between machines that are under local control.

### 3. A Quick Tour of the Uucp Software

Figure 1 is an illustration of the daemons used by the *uucp* network to communicate with another system. The *uucp(1)* or *uux(1)* command queues users requests and spawns the *uucico* daemon to call another system. Figure 2 illustrates the structure of *uucico* and the tasks that it performs in communicating with another system. It initiates the call to another system and performs the file transfer. On the receiving side, *uucico* is invoked to receive the transfer. Remote execution jobs are actually done by transferring a command file to the remote system and invoking a daemon (*uuxqt*) to execute that command file and return the results.

### 4. Installation

The *uucp(1)* package is delivered as part of the standard UNIX system distribution. It resides in its own subdirectory (called *uucp*) in the commands area and has its own make file (*uucp.mk*). The *uucp* package is installed as part of the normal distribution, however, if it must be reinstalled for any reason, then the sequence

```
make uucp.mk install
```

should be executed.

#### 4.1 Object Modules

The following object modules are installed as part of the *uucp* make procedure,

1. *Uucp* - The file transfer command.
2. *Uux* - The remote execution command.
3. *Uucico* - The *uucp* network daemon.
4. *Uustat* - Network status command.
5. *Uuclean* - Cleanup command.
6. *Uusub* - The command for monitoring and creating a subnetwork.
7. *Uuxqt* - The remote execution daemon.
8. *Uudemon.day* - A shell procedure that is invoked each day to maintain the network. Shell scripts for execution each week (*uudemon.wk*) and each hour (*uudemon.hr*) are also distributed.

#### 4.2 Password File

To allow remote systems to call the local system, password entries must be made for any *uucp* logins. For example,

```
nuucp:zaaAA:6:1:45422-UUCP.Admin:/usr/spool/uucppublic:/usr/lib/uucp/uucico
```



Note that the *uucico* daemon is used for the *shell* and the spool directory is used as the working directory.

#### 4.3 Lines File

The file */usr/lib/uucp/L-devices* contains the list of all lines that are directly connected to other systems or are available for calling other systems. The file contains the attributes of the lines and whether the line is a permanent connection or can call via a dialer. The format of the file is

*type line call-device speed protocol*

where each field is

- |                    |                                                                                                                                                                                                                                                                |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>type</i>        | Two keywords are used to describe whether a line is directly connected to another system (DIR) or uses an automatic calling unit (ACU). An X.25 permanent virtual circuit would use the DIR keyword.                                                           |
| <i>line</i>        | This is the device name for the line (e.g., <i>ttyab</i> for a direct line, <i>cul0</i> for a line connected to an ACU).                                                                                                                                       |
| <i>call-device</i> | If the ACU keyword is specified, this field contains the device name of the automatic calling unit. Otherwise, the field is ignored, however, a placeholder must be used in this field so that the <i>protocol</i> field can be interpreted.                   |
| <i>speed</i>       | The line speed that the connection is to run at. (The speed field is currently ignored if an X.25 link is used.)                                                                                                                                               |
| <i>protocol</i>    | This is an optional field that needs only be filled in if the connection is for a protocol other than the default terminal protocol. The X.25 protocol is the only other protocol supported and the single character <i>x</i> is used to select this protocol. |

The following entries illustrate various types of connections,

```
DIR ttyab 0 9600
ACU cul0 cua0 1200
DIR x25.s0 0 300 x
```

The first entry is for a hardwired line running at 9600 baud between two systems. Note that the *acu-device* field is zero. The second entry is for a line with a 1200 baud automatic calling unit. The last entry is for an X.25 synchronous direct connection between systems. Note that the *protocol* field is filled in and that the *acu-device* and *line speed* fields are meaningless.

**4.3.1 Naming Conventions** It is often useful when naming lines that are directly connected between systems or which are dedicated to calling other systems to choose a naming scheme that conveys the use of the line. In the earlier examples, the name *ttyab* is used for the line that directly connects two systems named *a* and *b*. Similarly, lines associated with calling units are best given names that relate them to the their calling unit (note the names *cul0* and *cua0* to specify the line and calling unit respectively).

#### 4.4 System File

Each entry in this file represents a system that can be called by the local *uucp* programs. More than one line may be present for a particular system. In this case, the additional lines represent alternative communication paths that will be tried in sequential order. The fields are described below.

*system name* The name of the remote system.



*time* This is a string that indicates the days-of-week and times-of-day when the system should be called (e.g., MoTuTh0800-1730).

The day portion may be a list containing some of *Su Mo Tu We Th Fr Sa* or it may be *Wk* for any week-day or *Any* for any day. The time should be a range of times (e.g., 0800-1230). If no time portion is specified, any time of day is assumed to be allowed for the call. Note that a time range that spans 0000 is permitted, for example, 0800-0600 means all times are allowed other than times between 6 and 8 am. An optional subfield is available to specify the minimum time (minutes) before a retry following a failed attempt. The subfield separator is a "," (e.g., Any,9 means call any time but wait at least 9 minutes before retrying the call after a failure has occurred).

*device* This is either *ACU* or the hard-wired device name to be used for the call. For the hard-wired case, the last part of the special file name is used (e.g., tty0).

*class* This is usually the line speed for the call (e.g., 300).

*phone* The phone number is made up of an optional alphabetic abbreviation (dialing prefix) and a numeric part. The abbreviation should be one that appears in the *L-dialcodes* file (e.g., mh5900, boston995-9980). For the hard-wired devices, this field contains the same string as used for the *device* field (e.g., tty0, etc.).

*login* The login information is given as a series of fields and subfields in the format

[ expect send ] ...

where *expect* is the string expected to be read and *send* is the string to be sent when the *expect* string is received.

The expect field may be made up of subfields of the form

expect[-send-expect] ...

where the *send* is sent if the prior *expect* is *not* successfully read and the *expect* following the *send* is the next expected string. (e.g., login--login will expect *login*; if it gets it, the program will go on to the next field; if it does not get *login*, it will send *null* followed by a new line, then expect *login* again.)

There are two special names available to be sent during the login sequence. The string *EOT* will send an EOT character and the string *BREAK* will try to send a *BREAK* character. (The *BREAK* character is simulated using line speed changes and null characters and may not work on all devices and/or systems.) A number from 1 to 9 may follow the *BREAK* for example, *BREAK1*, will send 1 null character instead of the default of 3. Note that *BREAK1* usually works best for 300/1200 baud lines.

A typical entry in the L.sys file would be

sys Any ACU 300 mh7654 login uucp ssword: word

The expect algorithm matches all or part of the input string as illustrated in the password field above.



#### 4.5 Dialing Prefixes

This file contains the dial-code abbreviations used in the *L.sys* file (e.g., py, mh, boston). The entry format is

abb dial-seq

where

abb is the abbreviation,

dial-seq is the dial sequence to call that location.

The line

py 165-

would be set up so that entry py7777 would send 165-7777 to the dial-unit.

#### 4.6 USERFILE

This file contains user accessibility information. It specifies four types of constraint,

- [1] which files can be accessed by a normal user of the local machine,
- [2] which files can be accessed from a remote computer,
- [3] which login name is used by a particular remote computer,
- [4] whether a remote computer should be called back in order to confirm its identity.

Each line in the file has the format

login,sys [ c ] path-name [ path-name ] ...

where

login is the login name for a user or the remote computer,

sys is the system name for a remote computer,

c is the optional *call-back required* flag,

path-name is a path-name prefix that is acceptable for sys.

The constraints are implemented as follows.

- [1] When the program is obeying a command stored on the local machine, the path-names allowed are those given on the first line in the *USERFILE* that has the login name of the user who entered the command. If no such line is found, the first line with a *null* login name is used.
- [2] When the program is responding to a command from a remote machine, the path-names allowed are those given on the first line in the file that has the system name that matches the remote machine. If no such line is found, the first one with a *null* system name is used.
- [3] When a remote computer logs in, the login name that it uses *must* appear in the *USERFILE*. There may be several lines with the same login name but one of



them must either have the name of the remote system or must contain a *null* system name.

- [4] If the line matched in ([3]) contains a "c", the remote machine is called back before any transactions take place.

The line

```
u,m /usr/xyz
```

allows machine *m* to login with name *u* and request the transfer of files whose names start with "/usr/xyz". The line

```
you, /usr/you
```

allows the ordinary user *you* to issue commands for files whose name starts with "/usr/you". (Note that this type restriction is seldom used.) The lines

```
u,m /usr/xyz /usr/spool
u, /usr/spool
```

allows *any* remote machine to login with name *u*. If its system name is not *m*, it can only ask to transfer files whose names start with "/usr/spool". If it is system *m*, it can send files from paths "/usr/xyz" as well as "/usr/spool". The lines

```
root, /
, /usr
```

allow any user to transfer files beginning with "/usr" but the user with login *root* can transfer any file. (Note that any file that is to be transferred must be readable by anybody.)

#### 4.7 Forwarding File

There are two files that allow restrictions to be placed on the forwarding mechanism. The format of the entries in each file is the same,

```
system
```

or

```
system!user!user2,...
```

The file *ORIGFILE* (*/usr/lib/uucp/ORIGFILE*) restricts the access of systems that are attempting to forward *through the local system*. The file contains the list of systems (and users) for whom the local system is willing to forward. Each entry refers to the system that was the *source* of the original job and not the name of the last system to forward the file. The second file *FWDFILE* (*/usr/lib/uucp/FWDFILE*) is a list of valid systems that a job can be *forwarded to* (it is not necessarily the name of the destination of a job, but merely the next valid node). This file will be a subset of the *L.sys* file and can be used to prevent forwarding to systems that are very expensive to reach, but to which access by local users is allowed (for example, links to overseas universities). If neither of these files exist, *uucp* will be perfectly happy to forward for any system. As an example, if the entry for system *australia* were in the *ORIGFILE* but not in the *FWDFILE* on system *xnode*, it would mean that system *australia* would be capable of forwarding jobs into the network via system *xnode* however, no systems in the network could forward a job to *australia* via system *xnode*.



## 5. Administration

The role of the *uucp* administrator depends heavily on the amount of traffic that enters or leaves a system and the quality of the connections that can be made to and from that system. For the average system, only a modest amount of traffic (100-200 files per day) pass through the system and little if any intervention with the *uucp* automatic cleanup functions is necessary. Systems that pass large numbers of files (200-10000) may require more attention when problems occur. The following sections describe the routine administrative tasks that must be performed by the administrator or are automatically performed by the *uucp* package. The section on problems describes what are the most frequent problems and how to effectively deal with them.

### 5.1 Cleanup

The biggest problem in a dialup network like *uucp* is dealing with the backlog of jobs that cannot be transmitted to other systems. The following cleanup activities should be routinely performed by shell scripts started from *cron(1)*.

**5.1.1 Cleanup of Undeliverable Jobs** The *uudemon.day* procedure usually contains an invocation of the *uuclean* command to purge any jobs that are older than some fixed time (usually 72 hours). A similar procedure is usually used to purge any *lock* or *status* files. An example invocation of *uuclean(1m)* to remove both job files and old status files every 48 hours is

```
usr/lib/uuclean -pST -pC -n48
```

**5.1.2 Cleanup of the Public Area** In order to keep the local filesystem from overflowing when files are sent to the public area, the *uudemon.day* procedure is usually set up with a *find* command to remove any files that are older than seven days. This interval may need to be shortened if there is not sufficient space to devote to the public area.

**5.1.3 Compaction of Log Files** The files *SYSLOG* and *LOGFILE* that contain logging information are compacted daily (using the *pack* command) and should be kept for one week before being overwritten.

### 5.2 Polling Other Systems

Systems that are passive members of the network must be polled by other systems in order for their files to be sent. This can be arranged by using the *uusub(1)* command as follows,

```
uusub -ccnode
```

which will call *cnode* when it is invoked.

### 5.3 Problems

The following sections list the most frequent problems that appear on systems that make heavy use of *uucp(1)*.

**5.3.1 Out of Space** The filesystem used to spool incoming or outgoing jobs can run out of space and prevent jobs from being spawned or much worse received from remote systems. The inability to receive jobs is the worse of the two conditions since when filespace does become available, the system will be *flooded* with the backlog of traffic.

**5.3.2 Bad CUs and Modems** The automatic calling units and incoming modems occasionally cause problems that make it difficult to contact other systems or to receive files. These problems are usually readily identifiable since *LOGFILE* entries will usually point to the bad line. If a bad line is suspected, it is useful to use the *cu(1)* command to try calling another system using the suspected line.



*5.3.3 Administrative problems* Some *uucp* networks have so many members that it is difficult to keep track of changing passwords, changing phone numbers or changing logins on remote systems. This can be a very costly problem since acu's will be tied up calling a system that cannot be reached.

## 6. Debugging

In order to verify that a system on the network can be contacted, the *uucico* daemon can be invoked from a user's terminal directly. For example, to verify that *cnode* can be contacted, a job would be queued for that system as follows,

```
uucp -r file cnode!~hom
```

The *-r* option forces the job to be queued but does not invoke the daemon to process the job. The *uucico* command can then be invoked directly,

```
lsr/libhuucphuucico -r1 -x4 -scnode
```

The *-r1* option is necessary to indicate that the daemon is to start up in *master* mode (that is, it is the calling system). The *-x4* specifies the level of debugging that is to be printed. Higher levels of debugging can be printed (greater than 4) but requires familiarity with the internals of *uucico*. If several jobs are queued for the remote system, it is not possible to force *uucico* to send one particular job first. The contents of LOGFILE should also be monitored for any error indications that it posts. Frequently, problems can be isolated by examining the entries in LOGFILE associated with a particular system. The file *ERRLOG* also contains error indications.

## 7. Conclusion

This manual has emphasized the format of control files and some of the issues in setting up a *uucp* network.



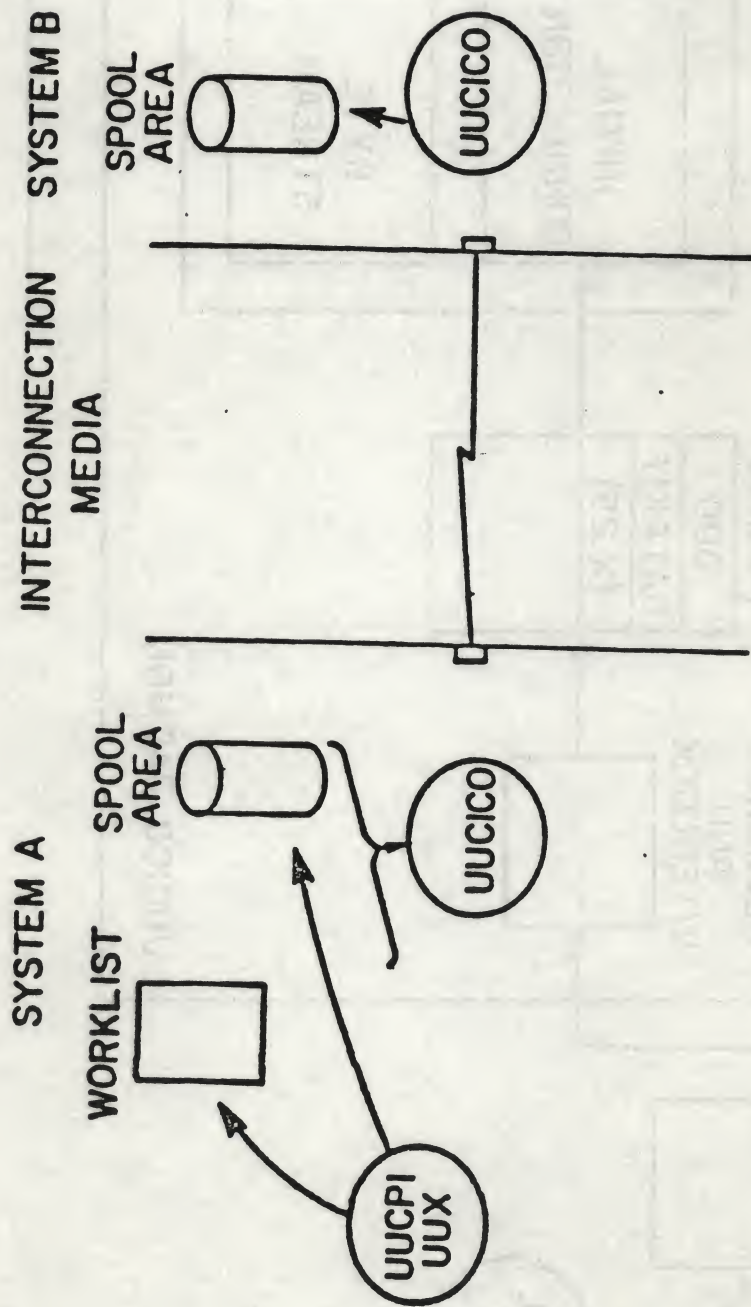


Figure 1 Uucp network daemons.



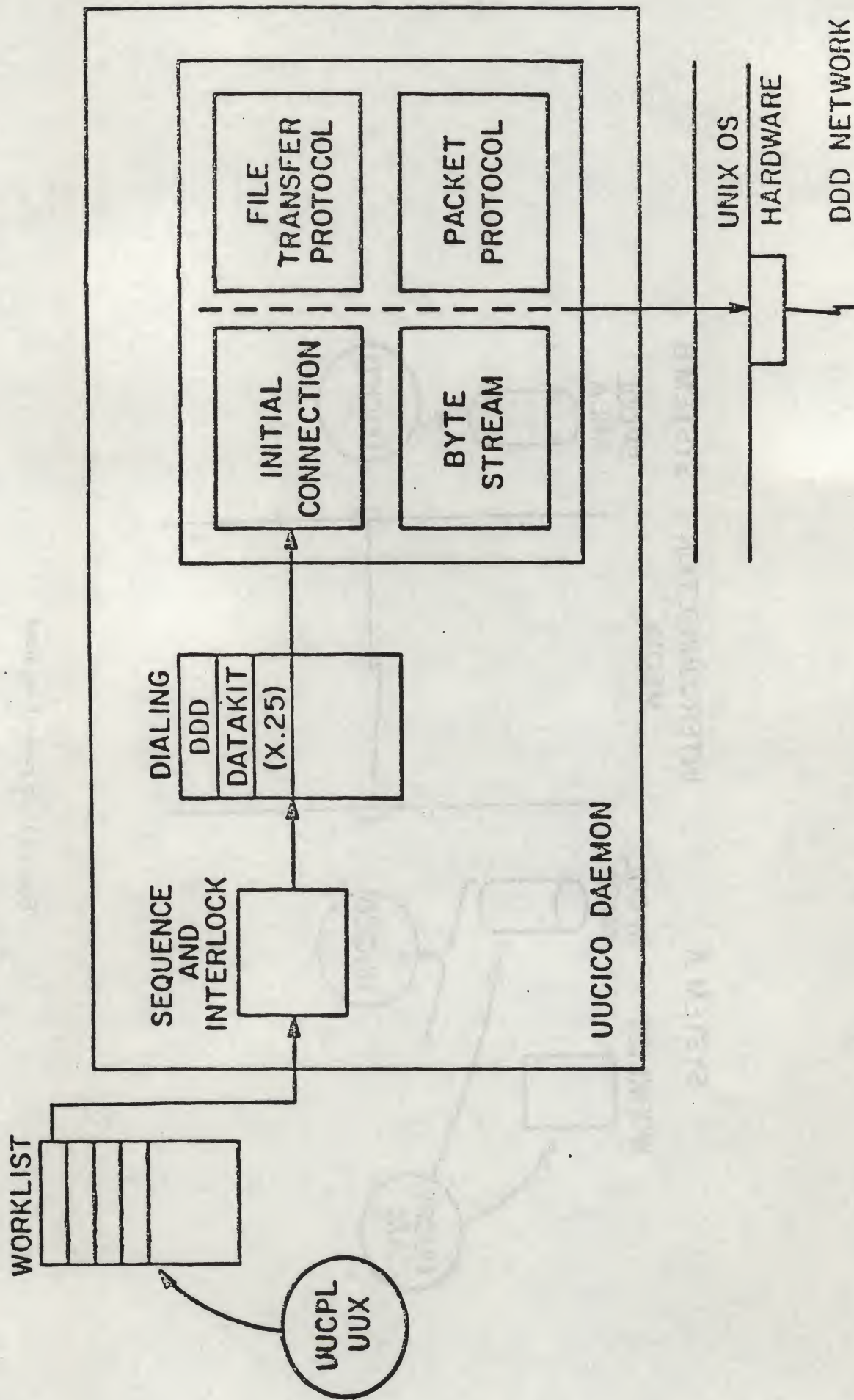


Figure 2 Uucico daemon functional blocks.



# UNIX System Accounting



This document is part of the ADMINISTRATOR'S GUIDE. Therefore the pagenumbers don't begin with 1.

**Trademarks:**

MUNIX, CADMUS  
DEC, PDP  
UNIX

for PCS  
for DEC  
for Bell Laboratories

Copyright 1984 by

PCS GmbH, Pfälzer-Wald-Strasse 36, D-8000 München 90, tel. (089) 67804-0

The information contained herein is the property of PCS and shall neither be reproduced in whole or in part without PCS's prior written approval nor be implied to grant any license to make, use or sell equipment manufactured herewith.

PCS reserves the right to make changes without notice in the specifications and materials contained herein and shall not be responsible for any damages (including consequential) caused by reliance on the materials presented.



## 7. UNIX SYSTEM ACCOUNTING

The UNIX System Accounting provides methods to collect per-process resource utilization data, record connect sessions, monitor disk utilization, and charge fees to specific logins. A set of C language programs and shell procedures is provided to reduce this accounting data into summary files and reports. This section describes the structure, implementation, and management of this accounting system, as well as a discussion of the reports generated and the meaning of the columnar data.

### GENERAL

The following list is a synopsis of the actions of the accounting system:

- At process termination, the UNIX system kernel writes one record per process in */usr/adm/pacct* in the form of *acct.h*. (See Attachment 7.1 for a description of data files.)
- The **login** and **init** programs record connect sessions by writing records into */etc/wtmp*. Date changes, reboots, and shutdowns are also recorded in this file.
- The disk utilization program **acctdusg** breaks down disk usage by login.
- Fees for file restores, etc., can be charged to specific logins with the **chargefee** shell procedure.
- Each day the **runacct** shell procedure is executed via **cron** to reduce accounting data and produce summary files and reports. (See Attachment 7.2 for a sample report output.)
- The **monacct** procedure can be executed on a monthly or fiscal period basis. It saves and restarts summary files, generates a report, and cleans up the *sum* directory. These saved summary files could be used to charge users for UNIX system usage.

### FILES AND DIRECTORIES

The */usr/lib/acct* directory contains all of the C language programs and shell procedures necessary to run the accounting system. The *adm* login (currently user ID of four) is used by the accounting system and has the directory structure shown in Fig. 7.1.

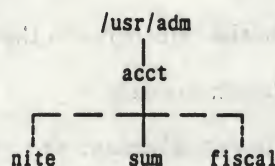


Fig. 7.1—Directory Structure of the "adm" Login

The */usr/adm* directory contains the active data collection files. (For a complete explanation of the files used by the accounting system, see Attachment 7.3.) The *nite* directory contains files that are reused daily by the **runacct** procedure. The *sum* directory contains the cumulative summary files updated by **runacct**. The *fiscal* directory contains periodic summary files created by **monacct**.



## DAILY OPERATION

When the UNIX system is switched into multiuser mode, `/usr/lib/acct/startup` is executed which does the following:

1. The `acctwtmp` program adds a "boot" record to `/usr/adm/wtmp`. This record is signified by using the system name as the login name in the `wtmp` record.
2. Process accounting is started via `turnacct`. `Turnacct on` executes the `accton` program with the argument `/usr/adm/pacct`.
3. The `remove` shell procedure is executed to clean up the saved `pacct` and `wtmp` files left in the `sum` directory by `runacct`.

The `ckpacct` procedure is run via `cron` every hour of the day to check the size of `/usr/adm/pacct`. If the file grows past 1000 blocks (default), `turnacct switch` is executed. While `ckpacct` is not absolutely necessary, the advantage of having several smaller `pacct` files becomes apparent when trying to restart `runacct` after a failure processing these records.

The `chargefee` program can be used to bill users for file restores, etc. It adds records to `/usr/adm/fee` which are picked up and processed by the next execution of `runacct` and merged into the total accounting records.

`Runacct` is executed via `cron` each night. It processes the active accounting files, `/usr/adm/pacct`, `/usr/adm/wtmp`, `/usr/adm/acct/nite/diskacct`, and `/usr/adm/fee`. It produces command summaries and usage summaries by login.

When the system is shut down using `shutdown`, the `shutacct` shell procedure is executed. It writes a shutdown reason record into `/usr/adm/wtmp` and turns process accounting off.

After the first reboot each morning, the computer operator should execute `/usr/lib/acct/prdaily` to print the previous day's accounting report.

## SETTING UP THE ACCOUNTING SYSTEM

In order to automate the operation of this accounting system, several things need to be done:

1. If not already present, add this line to the `/etc/rc` file in the state 2 section:

```
/bin/su -adm -c /usr/lib/acct/startup
```

2. If not already present, add this line to `/etc/shutdown` to turn off the accounting before the system is brought down:

```
/usr/lib/acct/shutacct
```

3. For most installations, the following three entries should be made in `/usr/lib/crontab` so that `cron` will automatically run the daily accounting.

```
" 0 4 * * 1-6 /bin/su -adm -c " /usr/lib/acct/runacct
    2> /usr/adm/acct/nite/fd2log "
0 2 * * 4 /usr/lib/acct/dodisk
5 * * * * /bin/su -adm -c " /usr/lib/acct/ckpacct "
```

Note that `dodisk` is invoked with superuser privileges of `root` so that directory searching is not road blocked.



4. To facilitate monthly merging of accounting data, the following entry in *crontab* will allow **monacct** to clean up all daily reports and daily total accounting files and deposit one monthly total report and one monthly total accounting file in the *fiscal* directory.

```
15 5 1 * * /bin/su -adm -c /usr/lib/acct/monacct
```

The above entry takes advantage of the default action of **monacct** that uses the current month's date as the suffix for the file names. Notice that the entry is executed at such a time as to allow **runacct** sufficient time to complete. This will, on the first day of each month, create monthly accounting files with the entire month's data.

5. The *PATH* shell variable should be set in */usr/adm/.profile* to:

```
PATH=/usr/lib/acct:/bin:/usr/bin
```

## RUNACCT

**Runacct** is the main daily accounting shell procedure. It is normally initiated via **cron** during nonprime time hours. **Runacct** processes connect, fee, disk, and process accounting files. It also prepares daily and cumulative summary files for use by **prdaily** or for billing purposes. The following files produced by **runacct** are of particular interest:

|               |                                                                                                                                                                                                                                                                                                                            |
|---------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| nite/lineuse  | Produced by <b>acctcon</b> , which reads the <i>wtmp</i> file, and produces usage statistics for each terminal line on the system. This report is especially useful for detecting bad lines. If the ratio between the number of logoffs to logins exceeds about 3/1, there is a good possibility that the line is failing. |
| nite/dayacct  | This file is the total accounting file for the previous day in <i>tacct.h</i> format.                                                                                                                                                                                                                                      |
| sum/tacct     | This file is the accumulation of each day's <i>nite/dayacct</i> which can be used for billing purposes. It is restarted each month or fiscal period by the <b>monacct</b> procedure.                                                                                                                                       |
| sum/daycms    | Produced by the <b>acctcms</b> program, it contains the daily command summary. The ASCII version of this file is <i>nite/daycms</i> .                                                                                                                                                                                      |
| sum/cms       | The accumulation of each day's command summaries. It is restarted by the execution of <b>monacct</b> . The ASCII version is <i>nite/cms</i> .                                                                                                                                                                              |
| sum/loginlog  | Produced by the <b>lastlogin</b> shell procedure, it maintains a record of the last time each login was used.                                                                                                                                                                                                              |
| sum/rprt.MMDD | Each execution of <b>runacct</b> saves a copy of the output of <b>prdaily</b> .                                                                                                                                                                                                                                            |

**Runacct** takes care not to damage files in the event of errors. A series of protection mechanisms are used that attempt to recognize an error, provide intelligent diagnostics, and terminate processing in such a way that **runacct** can be restarted with minimal intervention. It records its progress by writing descriptive messages into the file *active*. (Files used by **runacct** are assumed to be in the *nite* directory unless otherwise noted.) All diagnostics output during the execution of **runacct** is written into *fd2log*. To prevent multiple invocations, in the event of two **crons** or other problems, **runacct** will complain if the files *lock* and *lock1* exist when invoked. The *lastdate* file contains the month and day **runacct** was last invoked and is used to prevent more than one execution per day. If **runacct** detects an error, a message is written to */dev/console*, mail is sent to *root* and *adm*, the locks are removed, diagnostic files are saved, and execution is terminated.

In order to allow **runacct** to be restartable, processing is broken down into separate reentrant states. This is accomplished by using a **case** statement inside an endless **while** loop. Each state is one case of the **case**



statement. A file is used to remember the last state completed. When each state completes, *statefile* is updated to reflect the next state. In the next loop through the **while**, *statefile* is read and the **case** falls through to the next state. When **runacct** reaches the **CLEANUP** state, it removes the locks and terminates. States are executed as follows:

|            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SETUP      | The command <b>turnacct switch</b> is executed. The process accounting files, <i>/usr/adm/pacct?</i> , are moved to <i>/usr/adm/Spacct?.MMDD</i> . The <i>/usr/adm/wtmp</i> file is moved to <i>/usr/adm/acct/nite/wtmp.MMDD</i> with the current time added on the end.                                                                                                                                                                                                                                           |
| WTMPFIX    | The <i>wtmp</i> file in the <i>nite</i> directory is checked for correctness by the <b>wtmpfix</b> program. Some date changes will cause <b>acctcon1</b> to fail, so <b>wtmpfix</b> attempts to adjust the time stamps in the <i>wtmp</i> file if a date change record appears.                                                                                                                                                                                                                                    |
| CONNECT1   | Connect session records are written to <i>ctmp</i> in the form of <b>ctmp.h</b> . The <i>lineuse</i> file is created, and the <i>reboots</i> file is created showing all of the boot records found in the <i>wtmp</i> file.                                                                                                                                                                                                                                                                                        |
| CONNECT2   | <i>Ctmp</i> is converted to <i>ctacct.MMDD</i> which are connect accounting records. (Accounting records are in <b>tacct.h</b> format.)                                                                                                                                                                                                                                                                                                                                                                            |
| PROCESS    | The <b>acctprc1</b> and <b>acctprc2</b> programs are used to convert the process accounting files, <i>/usr/adm/Spacct?.MMDD</i> , into total accounting records in <i>ptacct?.MMDD</i> . The <i>Spacct</i> and <i>ptacct</i> files are correlated by number so that if <b>runacct</b> fails, the unnecessary reprocessing of <i>Spacct</i> files will not occur. One precaution should be noted; when restarting <b>runacct</b> in this state, remove the last <i>ptacct</i> file because it will not be complete. |
| MERGE      | Merge the process accounting records with the connect accounting records to form <i>daytacct</i> .                                                                                                                                                                                                                                                                                                                                                                                                                 |
| FEEES      | Merge in any ASCII <i>tacct</i> records from the file <i>fee</i> into <i>daytacct</i> .                                                                                                                                                                                                                                                                                                                                                                                                                            |
| DISK       | On the day after the <b>sdisk</b> procedure runs, merge <i>disktacct</i> with <i>daytacct</i> .                                                                                                                                                                                                                                                                                                                                                                                                                    |
| MERGETACCT | Merge <i>daytacct</i> with <i>sum/tacct</i> , the cumulative total accounting file. Each day, <i>daytacct</i> is saved in <i>sum/tacctMMDD</i> , so that <i>sum/tacct</i> can be recreated in the event it becomes corrupted or lost.                                                                                                                                                                                                                                                                              |
| CMS        | Merge in today's command summary with the cumulative command summary file <i>sum/cms</i> . Produce ASCII and internal format command summary files.                                                                                                                                                                                                                                                                                                                                                                |
| USEREXIT   | Any installation dependent (local) accounting programs can be included here.                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| CLEANUP    | Clean up temporary files, run <b>prdaily</b> and save its output in <i>sum/rprtMMDD</i> , remove the locks, then exit.                                                                                                                                                                                                                                                                                                                                                                                             |

#### RECOVERING FROM FAILURE

The **runacct** procedure can fail for a variety of reasons; usually due to a system crash, */usr* running out of space, or a corrupted *wtmp* file. If the *activeMMDD* file exists, check it first for error messages. If the *active* file and lock files exist, check *fd2log* for any mysterious messages. The following are error messages produced by **runacct**, and the recommended recovery actions:

ERROR: locks found, run aborted

The files *lock* and *lock1* were found. These files must be removed before **runacct** can restart.



ERROR: acctg already run for *date*: check /usr/adm/acct/nite/lastdate

The date in *lastdate* and today's date are the same. Remove *lastdate*.

ERROR: turnacct switch returned rc=?

Check the integrity of **turnacct** and **accton**. The **accton** program must be owned by *root* and have the *setuid* bit set.

ERROR: Spacct?.*MMDD* already exists

File setups probably already run. Check status of files, then run setups manually.

ERROR: /usr/adm/acct/nite/wtmp.*MMDD* already exists, run setup manually

Self-explanatory.

ERROR: wtmpfix errors see /usr/adm/acct/nite/wtmperror

**Wtmpfix** detected a corrupted *wtmp* file. Use **fwtmp** to correct the corrupted file.

ERROR: connect acctg failed: check /usr/adm/acct/nite/log

The **acctcon1** program encountered a bad *wtmp* file. Use **fwtmp** to correct the bad file.

ERROR: Invalid state, check /usr/adm/acct/nite/active

The file *statefile* is probably corrupted. Check *statefile* and read *active* before restarting.

#### RESTARTING RUNACCT

**Runacct** called without arguments assumes that this is the first invocation of the day. The argument *MMDD* is necessary if **runacct** is being restarted and specifies the month and day for which **runacct** will rerun the accounting. The entry point for processing is based on the contents of *statefile*. To override *statefile*, include the desired state on the command line. For example:

To start **runacct**:

```
nohup runacct 2> /usr/adm/acct/nite/fd2log&
```

To restart **runacct**:

```
nohup runacct 0601 2> /usr/adm/acct/nite/fd2log&
```

To restart **runacct** at a specific state:

```
nohup runacct 0601 WTMPFIX 2> /usr/adm/acct/nite/fd2log&
```

#### FIXING CORRUPTED FILES

Unfortunately, this accounting system is not entirely fool proof. Occasionally, a file will become corrupted or lost. Some of the files can simply be ignored or restored from the file save backup. However, certain files must be fixed in order to maintain the integrity of the accounting system.



### A. Fixing WTMP Errors

The *wtmp* files seem to cause the most problems in the day to day operation of the accounting system. When the date is changed and the UNIX system is in multiuser mode, a set of date change records is written into */usr/adm/wtmp*. The *wtmpfix* program is designed to adjust the time stamps in the *wtmp* records when a date change is encountered. Some combinations of date changes and reboots, however, will slip through *wtmpfix* and cause *acctcon1* to fail. The following steps show how to patch up a *wtmp* file.

```
cd /usr/adm/acct/nite
fwtmp < wtmp.MMDD > xwtmp
ed xwtmp
    delete corrupted records or
    delete all records from beginning up to the date change
fwtmp -ic < xwtmp > wtmp.MMDD
```

If the *wtmp* file is beyond repair, create a null *wtmp* file. This will prevent any charging of connect time. *Acctprcl* will not be able to determine which login owned a particular process, but it will be charged to the login that is first in the password file for that user id.

### B. Fixing TACCT Errors

If the installation is using the accounting system to charge users for system resources, the integrity of *sum/tacct* is quite important. Occasionally, mysterious *taacct* records will appear with negative numbers, duplicate user IDs, or a user ID of 65,535. First check *sum/tacctprev* with *prtacct*. If it looks all right, the latest *sum/tacct.MMDD* should be patched up, then *sum/tacct* recreated. A simple patchup procedure would be:

```
cd /usr/adm/acct/sum
acctmerg -v < tacct.MMDD > xtacct
ed xtacct
    remove the bad records
    write duplicate uid records to another file
acctmerg -i < xtacct > tacct.MMDD
acctmerg tacctprev < tacct.MMDD > tacct
```

Remember that the *monacct* procedure removes all the *taacct.MMDD* files; therefore, *sum/tacct* can be recreated by merging these files together.

### UPDATING PNPSPLIT

The *pnpsplit* subroutine is used by *acctcon1* and *acctprcl* to determine the difference between prime and nonprime time. Prime time is defaulted from 9:00 am to 5:00 pm, Monday through Friday. Nonprime time is considered to be all other hours and the entire day for those days listed in the *holidays* structure in *pnpsplit.c*. The holidays listed are accurate for Bell Laboratories New Jersey locations for the year the operating system was released. Every year on the day after Christmas (the last holiday of the calendar year), the following message will be printed on the system console terminal and appear in *log*.

\*\*\* RECOMPILE pnpsplit WITH NEW HOLIDAYS \*\*\*

This message will continue to be sent each time the accounting is run until *pnpsplit*, *acctcon1*, and *acctprcl* are recompiled. The following steps should be taken to successfully recompile these programs.

1. Edit *pnpsplit.c* to change the *thisyear* variable to the new year. Update the *holidays* structure to reflect the new holidays. The numeric entry in the structure is the day of the year, less one. For example, New Year's Day (January 1) is entered as 0. *Pnpsplit.c* is in */usr/src/cmd/acct/lib*.



2. Update the accounting library *a.a* and recompile **acctprcl**, and **acctconl** by:

superuser to root

ARGS= " acctconl acctprcl " /usr/src/:mkcmd acct

#### DAILY REPORTS

**Runacct** generates five basic reports upon each invocation. Samples of these reports are shown in Attachment 7.2. They cover the areas of connect accounting, usage by person on a daily basis, command usage reported by daily and monthly totals, and a report of the last time users were logged in.

The following paragraphs describe the reports and the meanings of their tabulated data.

##### A. Daily Report

In the first part of the report, the **from/to** banner should alert the administrator to the period reported on. The times are the time the last accounting report was generated until the time the current accounting report was generated. It is followed by a log of system reboots, shutdowns, power fail recoveries, and any other record dumped into */usr/adm/wtmp* by the **acctwtmp** program [see **acct(1M)** in the UNIX System Administrator's Manual].

The second part of the report is a breakdown of line utilization. The **TOTAL DURATION** tells how long the system was in multiuser state (able to be accessed through the terminal lines). The columns are:

|         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|---------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| LINE    | The terminal line or access port.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| MINUTES | The total number of minutes that line was in use during the accounting period.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| PERCENT | The total number of MINUTES the line was in use divided into the TOTAL DURATION.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| # SESS  | The number of times this port was accessed for a <b>login(1)</b> session.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| # ON    | This column does not have much meaning anymore. It used to give the number of times that the port was used to log a user on; but since <b>login(1)</b> can no longer be executed explicitly to log a new user in, this column should be identical with SESS.                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| # OFF   | This column reflects not just the number of times a user logged off but also any interrupts that occur on that line. Generally, interrupts occur on a port when the <b>getty(8)</b> is first invoked when the system is brought to multiuser state. These interrupts occur at a rate of about two per event; therefore, it is not uncommon to see in excess of twice the amount of OFF than ON or SESS. Where this column does come into play is when the # OFF exceeds the # ON by a large factor. This usually indicates that the multiplexer, modem or cable is going bad, or there is a bad connection somewhere. The most common cause of this is an unconnected cable dangling from the multiplexer. |

During real time, */usr/adm/wtmp* should be monitored as this is the file that the connect accounting is geared from. If it grows rapidly, execute **acctconl** to see which tty line is the most noisy. If the interrupting is occurring at a furious rate, general system performance will be effected.

##### B. Daily Usage Report

This report gives a by-user breakdown of system resource utilization. Its data consists of:

UID                      The user ID.



|                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| LOGIN NAME     | The login name of the user; there can be more than one login name for a single user ID, this identifies which one.                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| CPU (MINS)     | This represents the amount of time the user's process used the central processing unit. This category is broken down into PRIME and NPRIME (nonprime) utilization. The accounting system's idea of this breakdown is located in the accounting library function <code>pnpsplit</code> where the <code>holidays</code> array, which also determines nonprime time, is also defined. As delivered, prime time is defined to be 0900-1700 hours. The <code>holidays</code> array is correct for Bell Laboratories New Jersey locations for the year of the release.                      |
| KCORE-MINS     | This represents a cumulative measure of the amount of memory a process uses while running. The amount shown reflects kilobyte segments of memory used per minute. This measurement is also broken down into PRIME and NPRIME amounts.                                                                                                                                                                                                                                                                                                                                                 |
| CONNECT (MINS) | This identifies "Real Time" used. What this column really identifies is the amount of time that a user was logged into the system. If this time is rather high and the later column called # OF PROCS is low, this user is what is called a "line hog". That is, this person logs in first thing in the morning and does not hardly touch the terminal the rest of the day. Watch out for these kind of users. This column is also subdivided into PRIME and NPRIME utilization.                                                                                                      |
| DISK BLOCKS    | When the disk accounting programs have been run, their output is merged into the total accounting record ( <code>tacct.h</code> ) and shows up in this column. This disk accounting is accomplished by the program <code>acctdusg</code> .                                                                                                                                                                                                                                                                                                                                            |
| # OF PROCS     | This column reflects the number of processes that was invoked by the user. This is a good column to watch for large numbers indicating that a user may have a shell procedure that runs amock. The most common example of this is for a <code>crontab</code> entry to try to execute a user's <code>.profile</code> via <code>su-</code> that unfortunately prompts for a terminal type and sits in an endless loop trying to read from the terminal (there is not one when <code>cron</code> is executing a process). Preventive coding is encouraged in the <code>.profile</code> . |
| # OF SESS      | This is how many times the user logged onto the system.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| # DISK SAMPLES | This indicates how many times the disk accounting was run to obtain the average number of DISK BLOCKS listed earlier.                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| FEE            | An often unused field in the total accounting record, the FEE represents the total accumulation of widgets charged against the user by the <code>chargefee</code> shell procedure [see <code>acctsh(1M)</code> ]. The <code>chargefee</code> procedure is used to levy charges against a user for special services performed such as file restores, tape manipulation by operators, etc.                                                                                                                                                                                              |

### C. Daily Command and Monthly Total Command Summaries

These two reports are virtually the same except that the Daily Command Summary only reports on the current accounting period while the Monthly Total Command Summary tells the story for the start of the fiscal period to the current date. In other words, the monthly report reflects the data accumulated since the last invocation of `monacct`.

The data included in these reports gives an administrator an idea as to the heaviest used commands; and based on those commands' characteristics of system resource utilization, a hint as to what to weigh more heavily when system tuning.

These reports are sorted by TOTAL KCOREMIN which is an arbitrary yardstick, but often a good one for calculating "drain" on a system.



|                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| COMMAND NAME   | This is the name of the command. Unfortunately, all shell procedures are lumped together under the name <code>sh</code> since only object modules are reported by the process accounting system. The administrator should monitor the frequency of programs called <code>a.out</code> or <code>core</code> or any other name that does not seem quite right. Often people like to work on their favorite version of backgammon only they do not want everyone to know about it. <code>Acctcom</code> is also a good tool to use for determining who executed a suspiciously named command and also if superuser privileges were used. |
| NUMBER CMDS    | This is the total number of invocations of this particular command.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| TOTAL KCOREMIN | The total cumulative measurement of the amount of kilobyte segments of memory used by a process per minute of run time.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| TOTAL CPU-MIN  | The total processing time this program has accumulated.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| TOTAL REAL-MIN | The total real-time (wall-clock) minutes this program has accumulated. This total is the actual "waited for" time as opposed to kicking off a process in the background.                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| MEAN SIZE-K    | This is the mean of the TOTAL KCOREMIN over the number of invocations reflected by NUMBER CMDS.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| MEAN CPU-MIN   | This is the mean derived between the NUMBER CMDS and TOTAL CPU-MIN.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| HOG FACTOR     | This is a relative measurement of the ratio of system availability to system utilization. It is computed by the formula <div style="text-align: center;"> <math display="block">(\text{total CPU time}) / (\text{elapsed time})</math> </div> <p>This gives a relative measure of the total available CPU time consumed by the process during its execution.</p>                                                                                                                                                                                                                                                                      |
| CHARS TRNSFD   | This column, which may go negative, is a total count of the number of characters pushed around by the <code>read(2)</code> and <code>write(2)</code> system calls.                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| BLOCKS READ    | A total count of the physical block reads and writes that a process performed.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |

#### D. Last Login

This report simply gives the date when a particular login was last used. This could be a good source for finding likely candidates for the tape archives or getting rid of unused logins and login directories.

#### SUMMARY

The UNIX System Accounting was designed from a UNIX system administrator's point of view. Every possible precaution has been taken to ensure that the system will run smoothly and without error. It is important to become familiar with the C programs and shell procedures. The manual pages should be studied, and it is advisable to keep a printed copy of the shell procedures handy. The accounting system should be easy to maintain, provide valuable information for the administrator, and provide accurate breakdowns of the usage of system resources for charging purposes.



## ATTACHMENT 7.1

Format of wtmp files (utmp.h):

```

/*      %W%      */
/*      <sys/types.h> must be included.      */
#define UTMP_FILE      "/etc/utmp"
#define WTMP_FILE      "/etc/wtmp"
#define ut_name ut_user

struct utmp
{
    char ut_user[8];          /* User login name */
    char ut_id[4];           /* /etc/lines id(usually line #) */
    char ut_line[12];        /* device name (console, lnxx) */
    short ut_pid;            /* process id */
    short ut_type;           /* type of entry */
    struct exit_status
    {
        short e_termination; /* Process termination status */
        short e_exit;        /* Process exit status */
    }
    ut_exit;                /* The exit status of a process
                           * marked as DEAD_PROCESS.
                           */
    time_t ut_time;         /* time entry was made */
};

/*      Definitions for ut_type      */

#define EMPTY      0
#define RUN_LVL      1
#define BOOT_TIME      2
#define OLD_TIME      3
#define NEW_TIME      4
#define INIT_PROCESS      5      /* Process spawned by "init" */
#define LOGIN_PROCESS      6      /* A "getty" process waiting for login */
#define USER_PROCESS      7      /* A user process */
#define DEAD_PROCESS      8
#define ACCOUNTING      9

#define UTMAXTYPE      ACCOUNTING      /* Largest legal value of ut_type */

/*      Special strings or formats used in the "ut_line" field when
/*      accounting for something other than a process.
/*      No string for the ut_line field can be more than 11 chars +
/*      a NULL in length.

#define RUNLVL_MSG      "run-level %c".
#define BOOT_MSG      "system boot"
#define OTIME_MSG      "old time"
#define NTIME_MSG      "new time"

```



## ATTACHMENT 7.1 (Contd)

## Definitions (acctdef.h):

```

/*      %W% of %G%      */
/*
 *      defines, typedefs, etc. used by acct programs
 */

/*
 *      acct only typedefs
 */
typedef unsigned short  uid_t;

#ifdef u3b
#define HZ      100
#else
#define HZ      60
#endif

#define LSZ      12      /* sizeof line name */
#define NSZ      8       /* sizeof login name */
#define P        0       /* prime time */
#define NP       1       /* nonprime time */

/*
 *      limits which may have to be increased if systems get larger
 */
#define SSIZE     1000    /* max number of sessions in 1 acct run */
#define TSIZE     100     /* max number of line names in 1 acct run */
#define USIZE     500     /* max number of distinct login names in 1 acct run */

#define EQN(s1, s2)      (strcmp(s1, s2) == 0)
#define CPYN(s1, s2)     (strncmp(s1, s2, sizeof(s1)) == 0)

#define SECSINDAY      86400L
#define SECS(tics)     ((double) tics)/HZ
#define MINS(secs)     ((double) secs)/60
#define MINT(tics)     ((double) tics)/(60*HZ)

#ifdef pdp11
#define KCORE(clicks)  ((double) clicks/16)
#endif
#ifdef vax
#define KCORE(clicks)  ((double) clicks/2)
#endif
#ifdef u3b
#define KCORE(clicks)  ((double) clicks*2)
#endif

```



## ATTACHMENT 7.1 (Contd)

## Format of pacct files (acct.h):

```

/*
 * Accounting structures
 */

typedef ushort comp_t;      /* "floating point" */
                          /* 13-bit fraction, 3-bit exponent */

struct  acct

    char    ac_flag;        /* Accounting flag */
    char    ac_stat;        /* Exit status */
    ushort  ac_uid;         /* Accounting user ID */
    ushort  ac_gid;         /* Accounting group ID */
    dev_t   ac_tty;         /* control typewriter */
    time_t  ac_btime;       /* Beginning time */
    comp_t  ac_utime;       /* acctng user time in clock ticks */
    comp_t  ac_stime;       /* acctng system time in clock ticks */
    comp_t  ac_etime;       /* acctng elapsed time in clock ticks */
    comp_t  ac_mem;         /* memory usage */
    comp_t  ac_io;          /* chars transferred */
    comp_t  ac_rw;          /* blocks read or written */
    char    ac_comm[8];     /* command name */
;

extern struct acct  acctbuf;
extern struct inode *acctp; /* inode of accounting file */

#define AFORK      01      /* has executed fork, but no exec */
#define ASU        02      /* used superuser privileges */
#define ACCTF      0300    /* record type: 00 = acct */

```

## Format of tacct files (tacct.h):

```

/*
 * total acctounting (for acct period), also for day
 */

struct  tacct
    uid_t    ta_uid;        /* userid */
    char     ta_name[8];    /* login name */
    float    ta_cpu[2];     /* cum. cpu time, p/np (mins) */
    float    ta_kcore[2];   /* cum kcore-minutes, p/np */
    float    ta_con[2];     /* cum. connect time, p/np, mins */
    float    ta_du;         /* cum. disk usage */
    long     ta_pc;         /* count of processes */
    unsigned short ta_sc;    /* count of login sessions */
    unsigned short ta_dc;    /* count of disk samples */
    unsigned short ta_fee;   /* fee for special services */
;

```



## ATTACHMENT 7.1 (Contd)

Format of ctmp file (ctmp.h):

```
/*
 *   connect time record (various intermediate files)
 */
struct ctmp
    dev_t  ct_tty;      /*major minor */
    uid_t  ct_uid;      /*userid */
    char   ct_name[8];  /*login name */
    long   ct_con[2];   /*connect time (p/np) secs */
    time_t ct_start;    /*session start time */
;
```



## ATTACHMENT 7.2

Jun 8 04:14 1979 DAILY REPORT FOR pwba Page 1

from Thu Jun 7 06:00:48 1979

to Fri Jun 8 04:00:28 1979

2 shutdown

2 pwa

TOTAL DRATION IS 1320 MINUTES

| LINE    | MINUTES | PERCENT | / SESS | / ON | / OFF |
|---------|---------|---------|--------|------|-------|
| tty04   | 479     | 36      | 9      | 9    | 30    |
| tty47   | 341     | 26      | 4      | 4    | 33    |
| tty44   | 298     | 23      | 3      | 3    | 29    |
| tty46   | 336     | 25      | 9      | 9    | 33    |
| console | 1100    | 83      | 14     | 14   | 21    |
| tty05   | 448     | 34      | 3      | 3    | 22    |
| tty06   | 439     | 33      | 9      | 9    | 31    |
| tty07   | 421     | 32      | 6      | 6    | 24    |
| tty42   | 53      | 4       | 5      | 5    | 20    |
| tty09   | 385     | 29      | 11     | 11   | 33    |
| tty10   | 336     | 25      | 10     | 10   | 31    |
| tty08   | 464     | 35      | 2      | 2    | 19    |
| tty26   | 544     | 41      | 6      | 6    | 24    |
| tty12   | 252     | 19      | 5      | 5    | 25    |
| tty13   | 258     | 20      | 3      | 3    | 21    |
| tty14   | 156     | 12      | 6      | 6    | 26    |
| tty17   | 145     | 11      | 1      | 1    | 16    |
| tty18   | 39      | 3       | 5      | 5    | 24    |
| tty15   | 228     | 17      | 5      | 5    | 25    |
| tty25   | 704     | 53      | 6      | 6    | 25    |
| tty21   | 0       | 0       | 0      | 0    | 16    |
| tty19   | 10      | 1       | 1      | 1    | 17    |
| tty20   | 25      | 2       | 2      | 2    | 18    |
| tty22   | 0       | 0       | 0      | 0    | 15    |
| tty23   | 0       | 0       | 0      | 0    | 15    |
| tty24   | 0       | 0       | 0      | 0    | 16    |
| tty27   | 481     | 36      | 3      | 3    | 20    |
| tty28   | 426     | 32      | 5      | 5    | 24    |
| tty29   | 302     | 23      | 6      | 6    | 25    |
| tty30   | 257     | 20      | 11     | 11   | 28    |
| tty40   | 380     | 29      | 5      | 5    | 21    |
| tty41   | 343     | 26      | 3      | 3    | 21    |
| tty45   | 0       | 0       | 0      | 0    | 15    |
| tty11   | 365     | 28      | 7      | 7    | 25    |
| tty43   | 3       | 0       | 1      | 1    | 17    |
| tty16   | 213     | 16      | 3      | 3    | 20    |
| tty31   | 250     | 19      | 4      | 4    | 18    |
| tty02   | 62      | 5       | 1      | 1    | 3     |
| TOTALS  | 10544   | --      | 174    | 174  | 846   |



## ATTACHMENT 7.2 (Contd)

Jun 8 04:14 1979 DAILY USAGE REPORT FOR pwba Page 1

| UID  | LOGIN NAME | CPU (MINS) |        | KCORE-MINS |        | CONNECT (MINS) |        | DISK BLOCKS | # OF PROCS | # OF SESS | # DISK SAMPLES | FEE |
|------|------------|------------|--------|------------|--------|----------------|--------|-------------|------------|-----------|----------------|-----|
|      |            | PRIME      | NPRIME | PRIME      | NPRIME | PRIME          | NPRIME |             |            |           |                |     |
| 0    | TOTAL      | 388        | 103    | 12414      | 2934   | 9251           | 1056   | 0           | 16164      | 174       | 0              | 0   |
| 0    | root       | 47         | 41     | 1003       | 924    | 67             | 30     | 0           | 2360       | 8         | 0              | 0   |
| 4    | adm        | 27         | 19     | 48         | 652    | 0              | 0      | 0           | 842        | 0         | 0              | 0   |
| 19   | games      | 0          | 0      | 4          | 0      | 0              | 0      | 0           | 23         | 0         | 0              | 0   |
| 22   | mhb        | 0          | 0      | 1          | 1      | 1              | 1      | 0           | 14         | 2         | 0              | 0   |
| 37   | abs        | 0          | 0      | 4          | 0      | 0              | 0      | 0           | 3          | 0         | 0              | 0   |
| 37   | absjrk     | 14         | 0      | 284        | 0      | 423            | 0      | 0           | 1588       | 4         | 0              | 0   |
| 68   | rje        | 3          | 3      | 24         | 21     | 0              | 0      | 0           | 179        | 0         | 0              | 0   |
| 71   | ?          | 0          | 0      | 0          | 0      | 0              | 0      | 0           | 12         | 0         | 0              | 0   |
| 150  | jac        | 7          | 0      | 156        | 5      | 281            | 2      | 0           | 510        | 13        | 0              | 0   |
| 173  | ?          | 0          | 0      | 0          | 0      | 0              | 0      | 0           | 16         | 0         | 0              | 0   |
| 180  | ?          | 0          | 0      | 0          | 0      | 0              | 0      | 0           | 4          | 0         | 0              | 0   |
| 185  | ?          | 0          | 0      | 0          | 0      | 0              | 0      | 0           | 2          | 0         | 0              | 0   |
| 217  | denise     | 0          | 0      | 2          | 0      | 31             | 0      | 0           | 32         | 3         | 0              | 0   |
| 217  | kof        | 0          | 0      | 2          | 0      | 1              | 0      | 0           | 7          | 1         | 0              | 0   |
| 219  | ?          | 0          | 0      | 0          | 0      | 0              | 0      | 0           | 12         | 0         | 0              | 0   |
| 1001 | hsm        | 5          | 0      | 189        | 0      | 179            | 0      | 0           | 92         | 2         | 0              | 0   |
| 2001 | systst     | 0          | 1      | 5          | 23     | 476            | 64     | 0           | 99         | 5         | 0              | 0   |
| 2002 | mfp        | 1          | 0      | 7          | 5      | 270            | 62     | 0           | 93         | 3         | 0              | 0   |
| 2003 | als        | 1          | 0      | 23         | 0      | 100            | 0      | 0           | 99         | 3         | 0              | 0   |
| 2005 | eric       | 0          | 0      | 3          | 0      | 13             | 0      | 0           | 21         | 1         | 0              | 0   |
| 2006 | hoot       | 0          | 0      | 2          | 0      | 16             | 0      | 0           | 8          | 1         | 0              | 0   |
| 2009 | agp        | 47         | 0      | 2040       | 0      | 444            | 0      | 0           | 492        | 2         | 0              | 0   |
| 2009 | fsrepl     | 2          | 0      | 60         | 0      | 36             | 0      | 0           | 95         | 1         | 0              | 0   |
| 2011 | pdw        | 0          | 0      | 1          | 0      | 4              | 0      | 0           | 11         | 1         | 0              | 0   |
| 2012 | pwbat      | 0          | 0      | 1          | 0      | 23             | 0      | 0           | 9          | 1         | 0              | 0   |
| 2014 | cath       | 0          | 0      | 1          | 0      | 1              | 0      | 0           | 7          | 1         | 0              | 0   |
| 2022 | rem        | 32         | 1      | 1227       | 91     | 576            | 4      | 0           | 226        | 3         | 0              | 0   |
| 2025 | fld        | 55         | 23     | 2176       | 862    | 336            | 98     | 0           | 750        | 7         | 0              | 0   |
| 2027 | krb        | 14         | 2      | 365        | 51     | 547            | 24     | 0           | 372        | 8         | 0              | 0   |
| 2028 | text       | 0          | 0      | 1          | 0      | 3              | 0      | 0           | 13         | 1         | 0              | 0   |
| 2030 | arf        | 8          | 0      | 238        | 0      | 317            | 0      | 0           | 315        | 3         | 0              | 0   |
| 2031 | dp         | 12         | 0      | 480        | 3      | 459            | 6      | 0           | 220        | 6         | 0              | 0   |
| 2032 | graf       | 2          | 0      | 49         | 0      | 23             | 0      | 0           | 118        | 1         | 0              | 0   |
| 2033 | ecp        | 3          | 0      | 74         | 0      | 355            | 0      | 0           | 115        | 4         | 0              | 0   |
| 2040 | leap       | 15         | 0      | 308        | 0      | 513            | 1      | 0           | 505        | 2         | 0              | 0   |
| 2041 | dan        | 3          | 0      | 93         | 3      | 149            | 2      | 0           | 117        | 8         | 0              | 0   |
| 2051 | ds52       | 2          | 2      | 19         | 40     | 375            | 601    | 0           | 611        | 8         | 0              | 0   |
| 2055 | nuucp      | 0          | 0      | 15         | 9      | 17             | 1      | 0           | 10         | 3         | 0              | 0   |
| 2057 | ech        | 1          | 0      | 23         | 0      | 63             | 0      | 0           | 68         | 2         | 0              | 0   |
| 2061 | jcw        | 4          | 3      | 99         | 70     | 37             | 34     | 0           | 869        | 4         | 0              | 0   |
| 2064 | mjr        | 18         | 0      | 443        | 0      | 176            | 0      | 0           | 2065       | 3         | 0              | 0   |
| 2065 | rrr        | 0          | 0      | 6          | 0      | 7              | 0      | 0           | 23         | 1         | 0              | 0   |
| 2068 | trc        | 0          | 0      | 7          | 0      | 10             | 0      | 0           | 29         | 1         | 0              | 0   |
| 2075 | herb       | 29         | 0      | 1178       | 1      | 384            | 2      | 0           | 249        | 5         | 0              | 0   |
| 2086 | paul       | 1          | 0      | 14         | 0      | 152            | 0      | 0           | 28         | 1         | 0              | 0   |
| 2087 | pris       | 0          | 0      | 0          | 10     | 0              | 2      | 0           | 13         | 1         | 0              | 0   |
| 2111 | pwvcs      | 2          | 3      | 60         | 85     | 64             | 86     | 0           | 185        | 4         | 0              | 0   |
| 2116 | rbj        | 1          | 0      | 16         | 0      | 408            | 0      | 0           | 222        | 1         | 0              | 0   |
| 2121 | teach      | 0          | 0      | 3          | 0      | 53             | 0      | 0           | 50         | 2         | 0              | 0   |
| 2123 | msb        | 0          | 0      | 3          | 0      | 5              | 0      | 0           | 24         | 1         | 0              | 0   |
| 2124 | rnt        | 2          | 0      | 42         | 0      | 66             | 0      | 0           | 260        | 3         | 0              | 0   |
| 2126 | dai        | 0          | 0      | 5          | 0      | 121            | 0      | 0           | 17         | 1         | 0              | 0   |
| 2127 | m2         | 15         | 0      | 495        | 11     | 390            | 2      | 0           | 602        | 10        | 0              | 0   |

Jun 8 04:14 1979 DAILY USAGE REPORT FOR pwba Page 2

|      |         |    |   |     |    |     |    |   |     |   |   |   |
|------|---------|----|---|-----|----|-----|----|---|-----|---|---|---|
| 2128 | jel     | 14 | 0 | 492 | 9  | 422 | 14 | 0 | 523 | 8 | 0 | 0 |
| 2130 | sl      | 0  | 0 | 5   | 1  | 16  | 0  | 0 | 42  | 2 | 0 | 0 |
| 2130 | s3      | 0  | 0 | 0   | 0  | 0   | 2  | 0 | 9   | 1 | 0 | 0 |
| 2135 | jfn     | 0  | 1 | 0   | 12 | 0   | 11 | 0 | 33  | 2 | 0 | 0 |
| 2136 | m2class | 0  | 0 | 5   | 0  | 2   | 0  | 0 | 18  | 1 | 0 | 0 |
| 2140 | star    | 4  | 0 | 213 | 12 | 90  | 3  | 0 | 170 | 7 | 0 | 0 |
| 2141 | reg     | 5  | 0 | 245 | 25 | 470 | 4  | 0 | 181 | 1 | 0 | 0 |
| 2199 | lle     | 0  | 0 | 1   | 0  | 10  | 0  | 0 | 7   | 1 | 0 | 0 |
| 2999 | stock   | 0  | 0 | 1   | 0  | 1   | 0  | 0 | 17  | 1 | 0 | 0 |
| 3001 | whm     | 5  | 0 | 93  | 0  | 253 | 0  | 0 | 414 | 3 | 0 | 0 |
| 3332 | vjf     | 0  | 0 | 4   | 0  | 8   | 0  | 0 | 39  | 1 | 0 | 0 |



## ATTACHMENT 7.2 (Contd)

Jun 8 04:07 1979 DAILY COMMAND SUMMARY Page 1

| COMMAND NAME | NUMBER CMDS | TOTAL KCOREMIN | TOTAL CPU-MIN | TOTAL REAL-MIN | MEAN SIZE-K | MEAN CPU-MIN | HOG FACTOR | CHARS TRNSFD | BLOCKS READ |
|--------------|-------------|----------------|---------------|----------------|-------------|--------------|------------|--------------|-------------|
| TOTALS       | 16164       | 15332.89       | 490.72        | 37463.98       | 31.25       | 003          | 0.01       | 322183844    | 1097670     |
| nroff        | 119         | 3958.68        | 93.21         | 569.83         | 42.47       | 0.78         | 0.16       | 67070052     | 130284      |
| troff        | 26          | 2483.38        | 51.63         | 342.70         | 48.10       | 1.99         | 0.15       | 37869304     | 48989       |
| xnroff       | 20          | 732.03         | 16.74         | 111.05         | 43.73       | 0.84         | 0.15       | 13885248     | 22659       |
| a.out        | 31          | 623.53         | 10.52         | 142.77         | 59.26       | 0.34         | 0.07       | 382435       | 2758        |
| egrep        | 185         | 574.83         | 13.96         | 34.53          | 41.18       | 0.08         | 0.40       | 170625       | 8249        |
| m2fins       | 232         | 555.79         | 9.93          | 155.11         | 55.96       | 0.04         | 0.06       | 6155937      | 30994       |
| cl           | 150         | 519.04         | 13.57         | 48.89          | 38.25       | 0.09         | 0.28       | 4285724      | 16032       |
| c0           | 165         | 413.10         | 9.19          | 35.16          | 44.93       | 0.06         | 0.26       | 3827309      | 12170       |
| m2edit       | 33          | 340.92         | 4.63          | 148.27         | 73.62       | 0.14         | 0.03       | 1074914      | 14492       |
| ld           | 87          | 317.38         | 7.94          | 38.48          | 39.97       | 0.09         | 0.21       | 17640896     | 45797       |
| acctcms      | 17          | 294.75         | 6.49          | 14.15          | 45.41       | 0.38         | 0.46       | 2525427      | 5515        |
| c2           | 112         | 289.69         | 9.13          | 34.61          | 31.72       | 0.08         | 0.26       | 3667050      | 9681        |
| sh           | 1834        | 276.98         | 26.77         | 20444.24       | 10.35       | 0.01         | 0.00       | 3496613      | 71979       |
| ed           | 524         | 253.13         | 14.46         | 2029.89        | 17.50       | 0.03         | 0.01       | 18058108     | 56039       |
| acctprel     | 3           | 231.28         | 6.67          | 19.45          | 34.67       | 2.22         | 0.34       | 2577344      | 2926        |
| du           | 145         | 219.35         | 19.91         | 39.08          | 11.02       | 0.14         | 0.51       | 716389       | 23695       |
| diff         | 49          | 175.53         | 6.04          | 25.78          | 29.05       | 0.12         | 0.23       | 3740887      | 11351       |
| get          | 151         | 152.96         | 4.28          | 25.23          | 35.74       | 0.03         | 0.17       | 3634042      | 24917       |
| adb          | 22          | 148.10         | 4.07          | 202.35         | 36.37       | 0.19         | 0.02       | 2313718      | 9813        |
| tbl          | 24          | 143.43         | 2.44          | 210.65         | 58.71       | 0.10         | 0.01       | 1536210      | 3433        |
| dd           | 9           | 139.24         | 10.15         | 51.05          | 13.72       | 1.13         | 0.20       | 26006848     | 294         |
| as2          | 155         | 129.33         | 9.82          | 42.25          | 13.17       | 0.06         | 0.23       | 10500835     | 30165       |
| sed          | 597         | 115.46         | 4.19          | 36.23          | 27.57       | 0.01         | 0.12       | 783825       | 24497       |
| ps           | 51          | 109.69         | 5.92          | 41.55          | 18.54       | 0.12         | 0.14       | 2278056      | 8310        |
| make         | 89          | 102.94         | 2.87          | 203.32         | 35.81       | 0.03         | 0.01       | 1018461      | 8664        |
| delta        | 25          | 90.23          | 2.27          | 17.80          | 39.70       | 0.09         | 0.13       | 2909269      | 9321        |
| cpx          | 172         | 89.37          | 2.69          | 11.32          | 33.19       | 0.02         | 0.24       | 3519054      | 12155       |
| fsck         | 16          | 86.94          | 1.30          | 10.57          | 66.85       | 0.08         | 0.12       | 27671849     | 2927        |
| find         | 52          | 86.64          | 5.05          | 63.87          | 17.15       | 0.10         | 0.08       | 565125       | 11161       |
| ls           | 706         | 82.47          | 5.78          | 62.85          | 14.26       | 0.01         | 0.09       | 1811882      | 29659       |
| xck          | 2           | 79.44          | 10.49         | 47.89          | 7.57        | 5.25         | 0.22       | 198016       | 21995       |
| awk          | 22          | 78.83          | 1.37          | 5.24           | 57.72       | 0.06         | 0.26       | 355466       | 3769        |
| uucico       | 60          | 75.55          | 1.42          | 632.50         | 53.27       | 0.02         | 0.00       | 398693       | 6377        |
| acctcom      | 9           | 75.21          | 2.81          | 11.49          | 26.75       | 0.31         | 0.24       | 1283776      | 3771        |
| echo         | 2814        | 66.10          | 7.08          | 91.80          | 9.33        | 0.00         | 0.08       | 168651       | 24253       |
| ged          | 3           | 57.27          | 0.82          | 7.51           | 70.16       | 0.27         | 0.11       | 51832        | 426         |
| dc           | 284         | 56.92          | 2.42          | 9.43           | 23.48       | 0.01         | 0.26       | 14283        | 20329       |
| 450          | 7           | 48.03          | 6.80          | 84.45          | 7.06        | 0.97         | 0.08       | 279451       | 1700        |
| cat          | 749         | 45.49          | 5.69          | 478.54         | 8.00        | 0.01         | 0.01       | 8959500      | 27903       |
| ntd          | 6           | 41.52          | 1.55          | 7.55           | 26.87       | 0.26         | 0.20       | 59888        | 478         |
| mail         | 202         | 39.95          | 2.05          | 532.98         | 19.53       | 0.01         | 0.00       | 427217       | 14377       |
| acctpre2     | 3           | 38.95          | 1.43          | 19.45          | 27.24       | 0.48         | 0.07       | 587336       | 87          |
| sort         | 94          | 38.72          | 1.09          | 9.73           | 35.41       | 0.01         | 0.11       | 375876       | 4433        |
| pr           | 104         | 34.89          | 2.47          | 214.50         | 14.10       | 0.02         | 0.01       | 1060989      | 6572        |
| haspmain     | 7           | 33.20          | 5.28          | 1244.54        | 6.29        | 0.75         | 0.00       | 63064        | 36635       |
| ex           | 17          | 31.69          | 0.62          | 41.04          | 50.97       | 0.04         | 0.02       | 514624       | 3593        |
| grep         | 213         | 28.73          | 2.98          | 21.01          | 9.64        | 0.01         | 0.14       | 2100229      | 14297       |



## ATTACHMENT 7.2 (Contd)

Jun 8 04:07 1979 MONTHLY TOTAL SUMMARY Page 1

| COMMAND<br>NAME | NUMBER<br>CMDS | TOTAL<br>KCOREMIN | TOTAL<br>CPU-MIN | TOTAL<br>REAL-MIN | MEAN<br>SIZE-K | MEAN<br>CPU-MIN | HOG<br>FACTOR | CHARS<br>TRNSFD | BLOCKS<br>READ |
|-----------------|----------------|-------------------|------------------|-------------------|----------------|-----------------|---------------|-----------------|----------------|
| TOTALS          | 553286         | 297698.78         | 10916.09         | 742924.94         | 27.27          | 0.02            | 0.01          | 820472546       | 26253312       |
| nroff           | 1687           | 44681.55          | 995.92           | 5737.25           | 44.86          | 0.59            | 0.17          | 613403153       | 1089180        |
| troff           | 1351           | 25692.15          | 583.69           | 4356.05           | 44.02          | 0.43            | 0.13          | 413163589       | 646243         |
| spellpro        | 6466           | 17298.41          | 294.16           | 1893.79           | 58.81          | 0.05            | 0.16          | 334572640       | 853901         |
| m2edit          | 654            | 13526.69          | 164.62           | 4238.58           | 82.17          | 0.25            | 0.04          | 54940426        | 427924         |
| xnroff          | 397            | 10408.44          | 203.72           | 1496.32           | 51.09          | 0.51            | 0.14          | 215221419       | 301967         |
| sort            | 7983           | 9292.34           | 226.01           | 2298.05           | 41.11          | 0.03            | 0.10          | 80108304        | 355963         |
| cl              | 6139           | 8949.86           | 236.45           | 861.09            | 37.85          | 0.04            | 0.27          | 79897995        | 489661         |
| ld              | 3244           | 8852.96           | 223.19           | 1128.09           | 39.67          | 0.07            | 0.20          | 493701995       | 1278119        |
| sed             | 53134          | 8126.71           | 313.85           | 2241.78           | 25.89          | 0.01            | 0.14          | 23035033        | 1692990        |
| m2find          | 2982           | 7984.45           | 140.18           | 1698.25           | 56.96          | 0.05            | 0.08          | 111330040       | 449604         |
| c0              | 6586           | 7866.42           | 185.16           | 725.47            | 42.49          | 0.03            | 0.26          | 72595655        | 389426         |
| ed              | 20083          | 7822.78           | 425.90           | 41898.18          | 18.37          | 0.02            | 0.01          | 483425634       | 1541326        |
| tbl             | 660            | 7766.69           | 113.95           | 2458.55           | 68.16          | 0.17            | 0.05          | 50760094        | 83887          |
| sh              | 40476          | 7499.67           | 635.00           | 383786.53         | 11.81          | 0.02            | 0.00          | 70525236        | 1421194        |
| du              | 1941           | 6730.54           | 553.04           | 1128.44           | 12.17          | 0.28            | 0.49          | 20848359        | 628324         |
| a.out           | 1483           | 5658.46           | 126.87           | 1868.87           | 44.60          | 0.09            | 0.07          | 16158675        | 80260          |
| egrep           | 4801           | 5573.51           | 139.86           | 460.25            | 39.85          | 0.03            | 0.30          | 6823696         | 237298         |
| lintl           | 793            | 5325.66           | 71.23            | 425.67            | 74.76          | 0.09            | 0.17          | 9599001         | 131592         |
| cat             | 21170          | 4657.53           | 236.59           | 4354.24           | 19.69          | 0.01            | 0.05          | 239180412       | 1023965        |
| acctprel        | 42             | 3837.84           | 110.88           | 291.34            | 34.61          | 2.64            | 0.38          | 43954136        | 61123          |
| c2              | 4067           | 3807.25           | 144.86           | 477.28            | 26.28          | 0.04            | 0.30          | 57519376        | 213521         |
| grep            | 21212          | 3204.86           | 300.44           | 2727.87           | 10.67          | 0.01            | 0.11          | 139340583       | 899415         |
| cpp             | 7469           | 3060.72           | 94.12            | 647.79            | 32.52          | 0.01            | 0.15          | 91471956        | 459882         |
| getty           | 35556          | 2948.71           | 853.53           | 101107.45         | 3.45           | 0.02            | 0.01          | 34704751        | 263866         |
| m2editD         | 83             | 2707.27           | 28.79            | 361.84            | 94.02          | 0.35            | 0.08          | 2852202         | 33949          |
| as2             | 6454           | 2698.74           | 218.96           | 910.59            | 12.33          | 0.03            | 0.24          | 213336016       | 705690         |
| make            | 1858           | 2449.10           | 64.69            | 4388.86           | 37.86          | 0.03            | 0.01          | 24116259        | 175544         |
| ps              | 1034           | 2384.14           | 128.29           | 1207.87           | 18.58          | 0.12            | 0.11          | 54873792        | 204172         |
| acctcms         | 294            | 2288.36           | 51.99            | 116.06            | 44.01          | 0.18            | 0.45          | 36124940        | 80523          |
| uucico          | 815            | 2226.75           | 40.42            | 11729.01          | 55.08          | 0.05            | 0.00          | 11086105        | 162558         |
| ls              | 18876          | 2170.01           | 152.76           | 1538.09           | 14.20          | 0.01            | 0.10          | 32418106        | 691028         |
| find            | 1705           | 2114.18           | 114.35           | 920.75            | 18.49          | 0.07            | 0.12          | 94631199        | 338600         |
| ged             | 72             | 2026.43           | 28.54            | 317.21            | 71.01          | 0.40            | 0.09          | 1648636         | 10374          |
| echo            | 84710          | 2018.23           | 190.14           | 1138.49           | 10.61          | 0.00            | 0.17          | 2926992         | 649200         |
| cpio            | 127            | 1956.60           | 77.03            | 391.45            | 25.40          | 0.61            | 0.20          | 190822346       | 296302         |
| maze            | 8              | 1620.42           | 44.80            | 128.25            | 36.17          | 5.60            | 0.35          | 120399          | 212            |
| mail            | 4735           | 1474.38           | 76.92            | 14262.62          | 19.17          | 0.02            | 0.01          | 25719618        | 463748         |
| get             | 1085           | 1358.03           | 37.59            | 234.97            | 36.13          | 0.03            | 0.16          | 31540008        | 178623         |
| acctcom         | 165            | 1253.99           | 47.06            | 339.34            | 26.64          | 0.29            | 0.14          | 57405662        | 68949          |
| yacc            | 58             | 1187.17           | 15.36            | 36.90             | 77.31          | 0.26            | 0.42          | 4096070         | 12093          |
| col             | 638            | 1064.40           | 49.01            | 2199.00           | 21.72          | 0.08            | 0.02          | 23835395        | 16903          |
| line            | 27184          | 1036.03           | 93.14            | 1941.33           | 11.12          | 0.00            | 0.05          | 925447          | 296142         |
| nroff1.2        | 29             | 909.83            | 17.71            | 56.97             | 51.38          | 0.61            | 0.31          | 11459920        | 18802          |
| delta           | 264            | 904.54            | 23.07            | 254.06            | 39.21          | 0.09            | 0.09          | 24219141        | 87164          |
| td              | 175            | 886.19            | 25.74            | 159.73            | 34.43          | 0.15            | 0.16          | 1990177         | 15792          |
| ar              | 1434           | 872.65            | 61.87            | 309.07            | 14.11          | 0.04            | 0.20          | 189858731       | 428871         |
| m2findD         | 144            | 864.29            | 12.54            | 344.13            | 68.94          | 0.09            | 0.04          | 1184947         | 28576          |
| rm              | 15319          | 857.97            | 85.65            | 754.20            | 10.02          | 0.01            | 0.11          | 453479          | 433903         |
| acctdusg        | 1              | 819.77            | 39.30            | 170.10            | 20.86          | 39.30           | 0.23          | 1812480         | 39744          |
| f77pass1        | 155            | 779.13            | 7.97             | 29.09             | 97.70          | 0.05            | 0.27          | 990027          | 34702          |
| diff            | 786            | 767.31            | 32.77            | 260.27            | 23.41          | 0.04            | 0.13          | 22940094        | 97214          |



## ATTACHMENT 7.2 (Contd)

Jun 8 04:07 1979 LAST LOGIN Page 1

|          |          |          |        |          |         |
|----------|----------|----------|--------|----------|---------|
| 00-00-00 | dii      | 00-00-00 | rudd   | 79-06-08 | adm     |
| 00-00-00 | absadm   | 00-00-00 | s10    | 79-06-08 | agp     |
| 00-00-00 | absafr   | 00-00-00 | s2     | 79-06-08 | ais     |
| 00-00-00 | absas    | 00-00-00 | s4     | 79-06-08 | arf     |
| 00-00-00 | absjcw   | 00-00-00 | s5     | 79-06-08 | cath    |
| 00-00-00 | abspvg   | 00-00-00 | s6     | 79-06-08 | dal     |
| 00-00-00 | abstbm   | 00-00-00 | s8     | 79-06-08 | dan     |
| 00-00-00 | adm94    | 00-00-00 | s9     | 79-06-08 | denise  |
| 00-00-00 | apb      | 00-00-00 | scbsa  | 79-06-08 | dp      |
| 00-00-00 | archive  | 00-00-00 | sjm    | 79-06-08 | ds52    |
| 00-00-00 | asc      | 00-00-00 | srb    | 79-06-08 | ech     |
| 00-00-00 | badt     | 00-00-00 | sys    | 79-06-08 | ecp     |
| 00-00-00 | btb      | 00-00-00 | tgp    | 79-06-08 | eric    |
| 00-00-00 | bvl      | 00-00-00 | tld    | 79-06-08 | fld     |
| 00-00-00 | bwk      | 00-00-00 | ussc   | 79-06-08 | fsrepl  |
| 00-00-00 | chicken  | 00-00-00 | uucpa  | 79-06-08 | games   |
| 00-00-00 | class    | 00-00-00 | uvac   | 79-06-08 | graf    |
| 00-00-00 | cleary   | 00-00-00 | vav    | 79-06-08 | herb    |
| 00-00-00 | cs       | 00-00-00 | wdr    | 79-06-08 | hoot    |
| 00-00-00 | dba      | 00-00-00 | willa  | 79-06-08 | hsm     |
| 00-00-00 | deby     | 00-00-00 | zooma  | 79-06-08 | jac     |
| 00-00-00 | dec      | 79-06-04 | dws    | 79-06-08 | jcw     |
| 00-00-00 | demo     | 79-06-04 | ewb    | 79-06-08 | jel     |
| 00-00-00 | dlt      | 79-06-04 | kas    | 79-06-08 | jfn     |
| 00-00-00 | dmr      | 79-06-04 | satx   | 79-06-08 | kof     |
| 00-00-00 | docs     | 79-06-04 | uucp   | 79-06-08 | krb     |
| 00-00-00 | dug      | 79-06-05 | bcm    | 79-06-08 | leap    |
| 00-00-00 | ellie    | 79-06-05 | lprem  | 79-06-08 | lle     |
| 00-00-00 | fsrep2   | 79-06-05 | s7     | 79-06-08 | m2      |
| 00-00-00 | gas      | 79-06-05 | secs   | 79-06-08 | m2class |
| 00-00-00 | graphics | 79-06-06 | conv   | 79-06-08 | mfp     |
| 00-00-00 | hfg      | 79-06-06 | dck    | 79-06-08 | mhb     |
| 00-00-00 | hfb      | 79-06-06 | dmt    | 79-06-08 | mjr     |
| 00-00-00 | inst     | 79-06-06 | emp    | 79-06-08 | msb     |
| 00-00-00 | jfm      | 79-06-06 | pah    | 79-06-08 | nuucp   |
| 00-00-00 | jrh      | 79-06-06 | sync   | 79-06-08 | paul    |
| 00-00-00 | ken      | 79-06-06 | tad    | 79-06-08 | pdw     |
| 00-00-00 | lco      | 79-06-07 | ams    | 79-06-08 | pris    |
| 00-00-00 | learn    | 79-06-07 | bin    | 79-06-08 | pwbcs   |
| 00-00-00 | lppdw    | 79-06-07 | dgd    | 79-06-08 | pwbst   |
| 00-00-00 | lrbb     | 79-06-07 | haight | 79-06-08 | rbj     |
| 00-00-00 | maj      | 79-06-07 | hasp   | 79-06-08 | reg     |
| 00-00-00 | mar      | 79-06-07 | jgw    | 79-06-08 | rem     |
| 00-00-00 | mash     | 79-06-07 | leb    | 79-06-08 | rje     |
| 00-00-00 | meq      | 79-06-07 | ljk    | 79-06-08 | rnt     |
| 00-00-00 | miff     | 79-06-07 | mep    | 79-06-08 | root    |
| 00-00-00 | mle      | 79-06-07 | nhg    | 79-06-08 | rrr     |
| 00-00-00 | mmr      | 79-06-07 | nws    | 79-06-08 | sl      |
| 00-00-00 | mpf      | 79-06-07 | qtroff | 79-06-08 | s3      |
| 00-00-00 | plan     | 79-06-07 | tbm    | 79-06-08 | star    |
| 00-00-00 | plum     | 79-06-07 | train  | 79-06-08 | stock   |
| 00-00-00 | pvg      | 79-06-07 | whr    | 79-06-08 | systst  |
| 00-00-00 | rakesh   | 79-06-07 | wwe    | 79-06-08 | teach   |
| 00-00-00 | rfg      | 79-06-08 | ?      | 79-06-08 | text    |
| 00-00-00 | ric      | 79-06-08 | abs    | 79-06-08 | trc     |
| 00-00-00 | rre      | 79-06-08 | absjrk | 79-06-08 | vjf     |
|          |          |          |        | 79-06-08 | whm     |



## ATTACHMENT 7.3

## Files in the /usr/adm directory:

|              |                                                                             |
|--------------|-----------------------------------------------------------------------------|
| diskdiag     | diagnostic output during the execution of disk accounting programs          |
| dtmp         | output from the <i>acctdusg</i> program                                     |
| fee          | output from the <i>chargefee</i> program, ASCII <i>tacct</i> records        |
| pacct        | active process accounting file                                              |
| pacct?       | process accounting files switched via <i>turnacct</i>                       |
| Spacct?.MMDD | process accounting files for <i>MMDD</i> during execution of <i>runacct</i> |
| wtmp         | active <i>wtmp</i> file for recording connect sessions                      |

## Files in the /usr/adm/acct/nite directory:

|               |                                                                                                                                                                |
|---------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| active        | used by <i>runacct</i> to record progress and print warning and error messages; <i>active MMDD</i> same as <i>active</i> after <i>runacct</i> detects an error |
| cms           | ASCII total command summary used by <i>prdaily</i>                                                                                                             |
| ctacct.MMDD   | connect accounting records in <i>tacct.h</i> format                                                                                                            |
| ctmp          | output of <i>acctconl</i> program, connect session records in <i>ctmp.h</i> format                                                                             |
| daycms        | ASCII daily command summary used by <i>prdaily</i>                                                                                                             |
| dayacct       | total accounting records for one day in <i>tacct.h</i> format                                                                                                  |
| disktacct     | disk accounting records in <i>tacct.h</i> format, created by <i>dodisk</i> procedure                                                                           |
| fd2log        | diagnostic output during execution of <i>runacct</i> (see <i>cron</i> entry)                                                                                   |
| lastdate      | last day <i>runacct</i> executed in <i>date +%m%d</i> format                                                                                                   |
| lock lock1    | used to control serial use of <i>runacct</i>                                                                                                                   |
| lineuse       | tty line usage report used by <i>prdaily</i>                                                                                                                   |
| log           | diagnostic output from <i>acctconl</i>                                                                                                                         |
| logMMDD       | same as <i>log</i> after <i>runacct</i> detects an error                                                                                                       |
| reboots       | contains beginning and ending dates from <i>wtmp</i> , and a listing of reboots                                                                                |
| statefile     | used to record current state during execution of <i>runacct</i>                                                                                                |
| tmpwtmp       | <i>wtmp</i> file corrected by <i>wtmpfix</i>                                                                                                                   |
| wtmperror     | place for <i>wtmpfix</i> error messages                                                                                                                        |
| wtmperrorMMDD | same as <i>wtmperror</i> after <i>runacct</i> detects an error                                                                                                 |
| wtmp.MMDD     | previous day's <i>wtmp</i> file                                                                                                                                |



## ATTACHMENT 7.3 (Contd)

## Files in the /usr/adm/acct/sum directory:

|            |                                                                                                   |
|------------|---------------------------------------------------------------------------------------------------|
| cms        | total command summary file for current fiscal in internal summary format                          |
| cmsprev    | command summary file without latest update                                                        |
| daycms     | command summary file for yesterday in internal summary format                                     |
| loginlog   | created by <b>lastlogin</b>                                                                       |
| pacct.MMDD | concatenated version of all pacct files for MMDD, removed after reboot by <b>remove</b> procedure |
| rprrt.MMDD | saved output of <b>prdaily</b> program                                                            |
| tacct      | cumulative total accounting file for current fiscal                                               |
| tacctprev  | same as <b>tacct</b> without latest update                                                        |
| tacct.MMDD | total accounting file for MMDD                                                                    |
| wtmp.MMDD  | saved copy of wtmp file for MMDD, removed after reboot by <b>remove</b> procedure                 |

## Files in the /usr/adm/acct/fiscal directory:

|          |                                                                    |
|----------|--------------------------------------------------------------------|
| cms?     | total command summary file for fiscal ? in internal summary format |
| fiscrpt? | report similar to <b>prdaily</b> for fiscal ?                      |
| tacct?   | total accounting file for fiscal ?                                 |



THE UNIVERSITY OF MICHIGAN LIBRARY  
ANN ARBOR, MICHIGAN 48106-1000

## File System Checking

File System Checking  
The file system checking utility checks the file system for errors and repairs them. It is run on the root file system and on any other file systems that are mounted on the system.

The file system checking utility is run on the root file system and on any other file systems that are mounted on the system. It checks the file system for errors and repairs them. The utility is run on the root file system and on any other file systems that are mounted on the system. It checks the file system for errors and repairs them. The utility is run on the root file system and on any other file systems that are mounted on the system. It checks the file system for errors and repairs them.



This document is part of the ADMINISTRATOR'S GUIDE. Therefore the pagenumbers don't begin with 1.

Administrators Guide

**Trademarks:**

MUNIX, CADMUS  
DEC, PDP  
UNIX

for PCS  
for DEC  
for Bell Laboratories

Copyright 1984 by  
PCS GmbH, Pfälzer-Wald-Strasse 36, D-8000 München 90, tel. (089) 67804-0

The information contained herein is the property of PCS and shall neither be reproduced in whole or in part without PCS's prior written approval nor be implied to grant any license to make, use or sell equipment manufactured herewith.

PCS reserves the right to make changes without notice in the specifications and materials contained herein and shall not be responsible for any damages (including consequential) caused by reliance on the materials presented.



## 8. FILE SYSTEM CHECKING

The File System Check Program (**fsck**) is an interactive file system check and repair program. **Fsck** uses the redundant structural information in the UNIX file system to perform several consistency checks. If an inconsistency is detected, it is reported to the operator, who may elect to fix or ignore each inconsistency. These inconsistencies result from the permanent interruption of the file system updates, which are performed every time a file is modified. **Fsck** is frequently able to repair corrupted file systems using procedures based upon the order in which the UNIX system honors these file system update requests.

The purpose of this section is to describe the normal updating of the file system, to discuss the possible causes of file system corruption, and to present the corrective actions implemented by **fsck**. Both the program and the interaction between the program and the operator are described.

Appendix 8.1 contains the **fsck** error conditions. The meaning of the various error conditions, possible responses, and related error conditions are explained.

### GENERAL

When a UNIX operating system is brought up, a consistency check of the file systems should always be performed. This precautionary measure helps to ensure a reliable environment for file storage on disk. If an inconsistency is discovered, corrective action must be taken.

The purpose of this section is to dispel the mystique surrounding file system inconsistencies. The section describes the 5.0 file system, the updating of the file system, and then describes file system corruption. Finally, the set of heuristically sound corrective actions used by **fsck** are presented.

### THE 5.0 FILE SYSTEM

#### A. Introduction

- The 5.0 file system features a larger internal block size. The block size increased from 512 bytes to 1024 bytes and increased the performance of I/O bound applications. The size of the internal system buffers also increased to 1024 bytes. For a 1024-byte block file system, data transfers to/from disk are in 1024-byte operations.

#### B. Description

A 512-byte block file system is still supported by the operating system and file system related commands. Both file system sizes are allowed to coexist by detecting the file system type as set in the superblock. At file system mounting time, the operating system checks the magic number and type fields in the superblock. This magic number is unique in the sense that it is unlikely to be matched by an old 512-byte file system. A magic number mismatch assumes an original 512-byte block. New 1024-byte block file systems should have the special magic number set in the superblock and type field specifying a 1024-byte block. These fields are set at file system creation time (*/etc/mkfs*). Also, new file systems with 512-byte blocks may be created. These will have the special magic number set and type field indicating a 512-byte block. These fields in the superblock are set at creation time (*/etc/omkfs*). Labelit will report the file system type.

No functional changes should be perceived by the user. File system related commands have changed internally to handle both types of file systems. These changes are transparent to the user; the user interface remains unchanged. Most commands still report in 512-byte block units.

The root file system will be distributed as a 1024-byte block file system. Users are encouraged to convert their old file systems to the larger size block. However, 512-byte block file systems are still acceptable.



### C. System Administrator Advice

Remember that system buffers are now 1024 bytes. When configuring the operating system, take into consideration that the same number of buffers as before will use more main memory. Weigh this against reducing the number of buffers, which reduces the cache hit ratio and degrades performance.

### UPDATE OF THE FILE SYSTEM

Every working day hundreds of files are created, modified, and removed. Every time a file is modified, the UNIX operating system performs a series of file system updates. These updates, when written on disk, yield a consistent file system. To understand what happens in the event of a permanent interruption in this sequence, it is important to understand the order in which the update requests were probably being honored. Knowing which pieces of information were probably written to the file system first, heuristic procedures can be developed to repair a corrupted file system.

There are five types of file system updates. These involve the superblock, inodes, indirect blocks, data blocks (directories and files), and free-list blocks.

#### A. Superblock

The superblock contains information about the size of the file system, the size of the inode list, part of the free-block list, the count of free blocks, the count of free inodes, and part of the free-inode list.

The superblock of a mounted file system (the root file system is always mounted) is written to the file system whenever the file system is unmounted or a **sync** command is issued.

#### B. Inodes

An inode contains information about the type of inode (directory, data, or special), the number of directory entries linked to the inode, the list of blocks claimed by the inode, and the size of the inode.

An inode is written to the file system upon closure of the file associated with the inode. (All "in" core blocks are also written to the file system upon issue of a **sync** system call.)

#### C. Indirect Blocks

There are three types of indirect blocks—single indirect, double indirect, and triple indirect. A single-indirect block contains a list of some of the block numbers claimed by an inode. Each one of the 128 entries in an indirect block is a data-block number. A double-indirect block contains a list of single-indirect block numbers. A triple-indirect block contains a list of double-indirect block numbers.

Indirect blocks are written to the file system whenever they have been modified and released by the operating system. More precisely, they are queued for eventual writing. Physical I/O is deferred until the buffer is needed by the UNIX system or a **sync** command is issued.

#### D. Data Blocks

A data block may contain file information or directory entries. Each directory entry consists of a file name and an inode number.

Data blocks are written to the file system whenever they have been modified and released by the operating system.



#### E. First Free-List Block

The superblock contains the first free-list block. The free-list blocks are a list of all blocks that are not allocated to the superblock, inodes, indirect blocks, or data blocks. Each free-list block contains a count of the number of entries in this free-list block, a pointer to the next free-list block, and a partial list of free blocks in the file system.

Free-list blocks are written to the file system whenever they have been modified and released by the operating system.

### CORRUPTION OF THE FILE SYSTEM

A file system can become corrupted in a variety of ways. The most common of these ways are improper shutdown procedures and hardware failures.

#### A. Improper System Shutdown and Startup

File systems may become corrupted when proper shutdown procedures are not observed, e.g., forgetting to sync the system prior to halting the CPU, physically write-protecting a mounted file system, or taking a mounted file system off-line.

File systems may become further corrupted if proper startup procedures are not observed, e.g., not checking a file system for inconsistencies and not repairing inconsistencies. Allowing a corrupted file system to be used (and, thus, to be modified further) can be disastrous.

#### B. Hardware Failure

Any piece of hardware can fail at any time. Failures can be as subtle as a bad block on a disk pack or as blatant as a nonfunctional disk-controller.

### DETECTION AND CORRECTION OF CORRUPTION

A quiescent file system (i.e., an unmounted system and not being written on) may be checked for structural integrity by performing consistency checks on the redundant data intrinsic to a file system. The redundant data is either read from the file system or computed from other known values. A quiescent state is important during the checking of a file system because of the multipass nature of the **fsck** program.

When an inconsistency is discovered, **fsck** reports the inconsistency for the operator to choose a corrective action.

Discussed in this part are how to discover inconsistencies (and possible corrective actions) for the superblock, the inodes, the indirect blocks, the data blocks containing directory entries, and the free-list blocks. These corrective actions can be performed interactively by the **fsck** command under control of the operator.

#### A. Superblock

One of the most common corrupted items is the superblock. The superblock is prone to corruption because every change to the file system's blocks or inodes modifies the superblock.

The superblock and its associated parts are most often corrupted when the computer is halted and the last command involving output to the file system was not a **sync** command.

The superblock can be checked for inconsistencies involving file-system size, inode-list size, free-block list, free-block count, and the free-inode count.



### File-System Size and Inode-List Size

The file-system size must be larger than the number of blocks used by the superblock and the number of blocks used by the list of inodes. The number of inodes must be less than 65,535. The file-system size and inode-list size are critical pieces of information to the **fsck** program. While there is no way to actually check these sizes, **fsck** can check for them being within reasonable bounds. All other checks of the file system depend on the correctness of these sizes.

### Free-Block List

The free-block list starts in the superblock and continues through the free-list blocks of the file system. Each free-list block can be checked for a list count out of range, for block numbers out of range, and for blocks already allocated within the file system. A check is made to see that all the blocks in the file system were found.

The first free-block list is in the superblock. **Fsck** checks the list count for a value of less than 0 or greater than 50. It also checks each block number for a value of less than the first data block in the file system or greater than the last block in the file system. Then it compares each block number to a list of already allocated blocks. If the free-list block pointer is nonzero, the next free-list block is read in and the process is repeated.

When all the blocks have been accounted for, a check is made to see if the number of blocks used by the free-block list plus the number of blocks claimed by the inodes equals the total number of blocks in the file system.

If anything is wrong with the free-block list, then **fsck** may rebuild the list, excluding all blocks in the list of allocated blocks.

### Free-Block Count

The superblock contains a count of the total number of free blocks within the file system. **Fsck** compares this count to the number of blocks it found free within the file system. If the counts do not agree, then **fsck** may replace the count in the superblock by the actual free-block count.

### Free-Inode Count

The superblock contains a count of the total number of free inodes within the file system. **Fsck** compares this count to the number of inodes it found free within the file system. If the counts do not agree, then **fsck** may replace the count in the superblock by the actual free-inode count.

## B. Inodes

An individual inode is not as likely to be corrupted as the superblock. However, because of the great number of active inodes, there is almost as likely a chance for corruption in the inode list as in the superblock.

The list of inodes is checked sequentially starting with inode 1 (there is no inode 0) and going to the last inode in the file system. Each inode can be checked for inconsistencies involving format and type, link count, duplicate blocks, bad blocks, and inode size.

### Format and Type.

Each inode contains a mode word. This mode word describes the type and state of the inode. Inodes may be one of four types:

- regular
- directory



- special block
- special character.

If an inode is not one of these types, then the inode has an illegal type. Inodes may be found in one of three states—unallocated, allocated, and neither unallocated nor allocated. This last state indicates an incorrectly formatted inode. An inode can get in this state if bad data is written into the inode list through, for example, a hardware failure. The only possible corrective action is for **fsck** to clear the inode.

#### Link Count

Contained in each inode is a count of the total number of directory entries linked to the inode. **Fsck** verifies the link count of each inode by traversing down the total directory structure, starting from the root directory, and calculating an actual link count for each inode.

If the stored link count is nonzero and the actual link count is zero, it means that no directory entry appears for the inode. If the stored and actual link counts are nonzero and unequal, a directory entry may have been added or removed without the inode being updated.

If the stored link count is nonzero and the actual link count is zero, **fsck** may link the disconnected file to the *lost+found* directory. If the stored and actual link counts are nonzero and unequal, **fsck** may replace the stored link count by the actual link count.

#### Duplicate Blocks

Contained in each inode is a list or pointers to lists (indirect blocks) of all the blocks claimed by the inode. **Fsck** compares each block number claimed by an inode to a list of already allocated blocks. If a block number is already claimed by another inode, the block number is added to a list of duplicate blocks. Otherwise, the list of allocated blocks is updated to include the block number. If there are any duplicate blocks, **fsck** will make a partial second pass of the inode list to find the inode of the duplicated block. This is necessary because without examining the files associated with these inodes for correct content there is not enough information available to decide which inode is corrupted and should be cleared. Most times, the inode with the earliest modify time is incorrect and should be cleared. This condition can occur by using a file system with blocks claimed by both the free-block list and by other parts of the file system.

If there is a large number of duplicate blocks in an inode, this may be due to an indirect block not being written to the file system. **Fsck** will prompt the operator to clear both inodes.

#### Bad Blocks

Contained in each inode is a list or pointer to lists of all the blocks claimed by the inode. **Fsck** checks each block number claimed by an inode for a value lower than that of the first data block or greater than the last block in the file system. If the block number is outside this range, the block number is a bad block number.

If there is a large number of bad blocks in an inode, this may be due to an indirect block not being written to the file system. **Fsck** will prompt the operator to clear both inodes.

#### Size Checks

Each inode contains a 32 bit (4-byte) size field. This size indicates the number of characters in the file associated with the inode. This size can be checked for inconsistencies, e.g., directory sizes that are not a multiple of 16 characters or the number of blocks actually used not matching that indicated by the inode size.

A directory inode within the UNIX file system has the directory bit on in the inode mode word. The directory size must be a multiple of 16 because a directory entry contains 16 bytes (2 bytes for the inode number and 14 bytes for the file or directory name).



**Fsck** will warn of such directory misalignment. This is only a warning because not enough information can be gathered to correct the misalignment.

A rough check of the consistency of the size field of an inode can be performed by computing from the size field the number of blocks that should be associated with the inode and comparing it to the actual number of blocks claimed by the inode.

**Fsck** calculates the number of blocks that there should be in an inode by dividing the number of characters in an inode by the number of characters per block and rounding up. **Fsck** adds one block for each indirect block associated with the inode. If the actual number of blocks does not match the computed number of blocks, **fsck** will warn of a possible file-size error. This is only a warning because the UNIX system does not fill in blocks in files created in random order.

### C. Indirect Blocks

Indirect blocks are owned by an inode. Therefore, inconsistencies in indirect blocks directly affect the inode that owns it.

Inconsistencies that can be checked are blocks already claimed by another inode and block numbers outside the range of the file system.

For a discussion of detection and correction of the inconsistencies associated with indirect blocks, see the parts "Duplicate Blocks" and "Bad Blocks".

### D. Data Blocks

The two types of data blocks are plain data blocks and directory data blocks. Plain data blocks contain the information stored in a file. Directory data blocks contain directory entries. **Fsck** does not attempt to check the validity of the contents of a plain data block.

Each directory data block can be checked for inconsistencies involving directory inode numbers pointing to unallocated inodes, directory inode numbers greater than the number of inodes in the file system, incorrect directory inode numbers for "." and "..", and directories which are disconnected from the file system. In addition, the validity of the contents of a directory's data block is checked.

If a directory entry inode number points to an unallocated inode, then **fsck** may remove that directory entry. This condition probably occurred because the data blocks containing the directory entries were modified and written out while the inode was not yet written out.

If a directory entry inode number is pointing beyond the end of the inode list, **fsck** may remove that directory entry. This condition occurs if bad data is written into a directory data block.

The directory inode number entry for "." should be the first entry in the directory data block. Its value should be equal to the inode number for the directory data block.

The directory inode number entry for ".." should be the second entry in the directory data block. Its value should be equal to the inode number for the parent of the directory entry (or the inode number of the directory data block if the directory is the root directory).

If the directory inode numbers are incorrect, **fsck** may replace them by the correct values.

**Fsck** checks the general connectivity of the file system. If directories are found not to be linked into the file system, **fsck** will link the directory back into the file system in the *lost+found* directory. This condition



can be caused by inodes being written to the file system with the corresponding directory data blocks not being written to the file system.

#### **E. Free-List Blocks**

Free-list blocks are owned by the superblock. Therefore, inconsistencies in free-list blocks directly affect the superblock.

Inconsistencies that can be checked are a list count outside of range, block numbers outside of range, and blocks already associated with the file system.

For a discussion of detection and correction of the inconsistencies associated with free-list blocks, see the part "Free-Block List".



## APPENDIX 8.1

## FSCK ERROR CONDITIONS

## A. Conventions

**Fsck** is a multipass file system check program. Each file system pass invokes a different phase of the **fsck** program. After the initial setup, **fsck** performs successive phases over each file system performing cleanup, checking blocks and sizes, pathnames, connectivity, reference counts, and the free-block list (possibly rebuilding it).

When an inconsistency is detected, **fsck** reports the error condition to the operator. If a response is required, **fsck** prints a prompt message and waits for a response. This appendix explains the meaning of each error condition, the possible responses, and the related error conditions.

The error conditions are organized by the "Phase" of the **fsck** program in which they can occur. The error conditions that may occur in more than one phase will be discussed under the part "Initialization".

## B. Initialization

Before a file system check can be performed, certain tables have to be set up and certain files opened. This part concerns itself with the opening of files and the initialization of tables. Error conditions resulting from command line options, memory requests, opening of files, status of files, file system size checks, and creation of the scratch file are listed below.

## C option?

C is not a legal option to **fsck**; legal options are **-y**, **-n**, **-s**, **-S**, **-t**, **-f**, **-q**, and **-D**. **Fsck** terminates on this error condition. See the **fsck(1M)** entry in the UNIX System Administrator's Manual for further details.

## Bad -t option

The **-t** option is not followed by a file name. **Fsck** terminates on this error condition. See the **fsck(1M)** entry in the UNIX System Administrator's Manual for further details.

## Invalid -s argument, defaults assumed

The **-s** option is not suffixed by 3, 4, or blocks-per-cylinder:blocks-to-skip. **Fsck** assumes a default value of 400 blocks-per-cylinder and 7 blocks-to-skip. See the **fsck(1M)** entry in the UNIX System Administrator's Manual for further details.

## Incompatible options: -n and -s

It is not possible to salvage the free-block list without modifying the file system. **Fsck** terminates on this error condition. See the **fsck(1M)** entry in the UNIX System Administrator's Manual for further details.

## Incompatible options: -n and -q

It is not possible to do automatic removal without modifying the file system. **Fsck** terminates on this error condition. See the **fsck(1M)** entry in the UNIX System Administrator's Manual for further details.

## Can not fstat standard input

**Fsck's** attempt to **fstat** standard input failed. This should never happen. **Fsck** terminates on this error condition.



**Can not get memory**

Fsck's request for memory for its virtual memory tables failed. This should never happen. Fsck terminates on this error condition.

**Can not open checklist file: F**

The default file system checklist file *F* (usually */etc/checklist*) can not be opened for reading. Fsck terminates on this error condition. Check access modes of *F*.

**Can not stat root**

Fsck's request for statistics about the root directory */* failed. This should never happen. Fsck terminates on this error condition.

**Can not stat F**

Fsck's request for statistics about the file system *F* failed. It ignores this file system and continues checking the next file system given. Check access modes of *F*.

**FS is a mounted file system, ignored**

This is to avoid modifying a mounted file system. It ignores this file system and continues with the next file system given.

**F is not a block or character device**

Fsck has been given a regular file name by mistake. It ignores this file system and continues checking the next file system given. Check file type of *F*.

**Can not open F**

The file system *F* can not be opened for reading. It ignores this file system and continues checking the next file system given. Check access modes of *F*.

**Size check: fsize X isize Y**

More blocks are used for the inode list *Y* than there are blocks in the file system *X*, or there are more than 65,535 inodes in the file system. It ignores this file system and continues checking the next file system given.

**Can not create F**

Fsck's request to create a scratch file *F* failed. It ignores this file system and continues checking the next file system given. Check access modes of *F*.

**CAN NOT SEEK: BLK B (CONTINUE)**

Fsck's request for moving to a specified block number *B* in the file system failed. This should never happen.

Possible responses to CONTINUE prompt are:

YES

Attempt to continue to run file system check. Often, however, the problem will persist. This error condition will not allow a complete check of the file system. A second run of



**fsck** should be made to recheck this file system. If block was part of the virtual memory buffer cache, **fsck** will terminate with the message "Fatal I/O error".

NO Terminate program.

**CAN NOT READ: BLK B (CONTINUE)**

**Fsck's** request for reading a specified block number *B* in the file system failed. This should never happen.

Possible responses to **CONTINUE** prompt are:

YES Attempt to continue to run file system check. Often, however, the problem will persist. This error condition will not allow a complete check of the file system. A second run of **fsck** should be made to recheck this file system. If block was part of the virtual memory buffer cache, **fsck** will terminate with the message "Fatal I/O error".

NO Terminate program.

**CAN NOT WRITE: BLK B (CONTINUE)**

**Fsck's** request for writing a specified block number *B* in the file system failed. The disk is write-protected.

Possible responses to **CONTINUE** prompt are:

YES Attempt to continue to run file system check. Often, however, the problem will persist. This error condition will not allow a complete check of the file system. A second run of **fsck** should be made to recheck this file system. If block was part of the virtual memory buffer cache, **fsck** will terminate with the message "Fatal I/O error".

NO Terminate program.

**C. PHASE 1: CHECK BLOCKS AND SIZES**

This phase concerns itself with the inode list. This part lists error conditions resulting from checking inode types, setting up the zero-link-count table, examining inode block numbers for bad or duplicate blocks, checking inode size, and checking inode format.

**UNKNOWN FILE TYPE I=I (CLEAR)**

The mode word of the inode *I* indicates that the inode is not a special character inode, special character inode, regular inode, or directory inode. See the part "Format and Types" for more information.

Possible responses to **CLEAR** prompt are:

YES Deallocate inode *I* by zeroing its contents. This will always invoke the **UNALLOCATED** error condition in Phase 2 for each directory entry pointing to this inode.

NO Ignore this error condition.

**LINK COUNT TABLE OVERFLOW (CONTINUE)**

An internal table for **fsck** containing allocated inodes with a link count of zero has no more room. Recompile **fsck** with a larger value of **MAXLNCNT**.



Possible responses to CONTINUE prompt are:

YES Continue with program. This error condition will not allow a complete check of the file system. A second run of **fsck** should be made to recheck this file system. If another allocated inode with a zero link count is found, this error condition is repeated.

NO Terminate program.

#### B BAD I=I

Inode *I* contains block number *B* with a number lower than the number of the first data block in the file system or greater than the number of the last block in the file system. This error condition may invoke the EXCESSIVE BAD BLKS error condition in Phase 1 if inode *I* has too many block numbers outside the file system range. This error condition will always invoke the BAD/DUP error condition in Phase 2 and Phase 4. See the part "Bad Blocks" for more information.

#### EXCESSIVE BAD BLKS I=I (CONTINUE)

There is more than a tolerable number (usually 10) of blocks with a number lower than the number of the first data block in the file system or greater than the number of last block in the file system associated with inode *I*. See the part "Bad Blocks" for more information.

Possible responses to CONTINUE prompt are:

YES Ignore the rest of the blocks in this inode and continue checking with next inode in the file system. This error condition will not allow a complete check of the file system. A second run of **fsck** should be made to recheck this file system.

NO Terminate program.

#### B DUP I=I

Inode *I* contains block number *B* which is already claimed by another inode. This error condition may invoke the EXCESSIVE DUP BLKS error condition in Phase 1 if inode *I* has too many block numbers claimed by other inodes. This error condition will always invoke Phase 1b and the BAD/DUP error condition in Phase 2 and Phase 4. See the part "Duplicate Blocks" for more information.

#### EXCESSIVE DUP BLKS I=I (CONTINUE)

There is more than a tolerable number (usually 10) of blocks claimed by other inodes. See the part "Duplicate Blocks" for more information.

Possible responses to CONTINUE prompt are:

YES Ignore the rest of the blocks in this inode and continue checking with next inode in the file system. This error condition will not allow a complete check of the file system. A second run of **fsck** should be made to recheck this file system.

NO Terminate program.

#### DUP TABLE OVERFLOW (CONTINUE)

An internal table in **fsck** containing duplicate block numbers has no more room. Recompile **fsck** with a larger value of DUPTBLSIZE.



Possible responses to CONTINUE prompt are:

- YES Continue with program. This error condition will not allow a complete check of the file system. A second run of `fsck` should be made to recheck this file system. If another duplicate block is found, this error condition will repeat.
- NO Terminate program.

#### POSSIBLE FILE SIZE ERROR I=I

The inode *I* size does not match the actual number of blocks used by the inode. This is only a warning. (See the part "Size Checks".) If the `-q` option is used, this message is not printed.

#### DIRECTORY MISALIGNED I=I

The size of a directory inode is not a multiple of the size of a directory entry (usually 16). This is only a warning. (See the part "Size Checks".) If the `-q` option is used, this message is not printed.

#### PARTIALLY ALLOCATED INODE I=I (CLEAR)

Inode *I* is neither allocated nor unallocated. See the part "Format and Types" for more information.

Possible responses to CLEAR prompt are:

- YES Deallocate inode *I* by zeroing its contents.
- NO Ignore this error condition.

#### D. PHASE 1B: RESCAN FOR MORE DUPS

When a duplicate block is found in the file system, the file system is rescanned to find the inode which previously claimed that block. This part lists the error condition when the duplicate block is found.

#### B DUP I=I

Inode *I* contains block number *B* which is already claimed by another inode. This error condition will always invoke the BAD/DUP error condition in Phase 2. Inodes with overlapping blocks may be determined by examining this error condition and the DUP error condition in Phase 1. See the part "Duplicate Blocks" for more information.

#### E. PHASE 2: CHECK PATHNAMES

This phase concerns itself with removing directory entries pointing to error conditioned inodes from Phase 1 and Phase 1b. This part lists error conditions resulting from root inode mode and status, directory inode pointers in range, and directory entries pointing to bad inodes.

#### ROOT INODE UNALLOCATED. TERMINATING

The root inode (always inode number 2) has no allocate mode bits. This should never happen. The program will terminate. See the part "Format and Types" for more information.

#### ROOT INODE NOT DIRECTORY (FIX)

The root inode (usually inode number 2) is not directory inode type.



Possible responses to FIX prompt are:

YES                      Replace the root inode's type to be a directory. If the root inode's data blocks are not directory blocks, a very large number of error conditions will be produced.

NO                        Terminate program.

**DUPS/BAD IN ROOT INODE (CONTINUE)**

Phase 1 or Phase 1b have found duplicate blocks or bad blocks in the root inode (usually inode number 2) for the file system.

Possible responses to CONTINUE prompt are:

YES                      Ignore DUPS/BAD error condition in root inode and attempt to continue to run the file system check. If root inode is not correct, then this may result in a large number of other error conditions.

NO                        Terminate program.

**I OUT OF RANGE I=I NAME=F (REMOVE)**

A directory entry *F* has an inode number *I* which is greater than the end of the inode list. See the part "Data Blocks" for more information.

Possible responses to REMOVE prompt are:

YES                      The directory entry *F* is removed.

NO                        Ignore this error condition.

**UNALLOCATED I=I OWNER=O MODE=M SIZE=S MTIME=T NAME=F (REMOVE)**

A directory entry *F* has an inode *I* without allocate mode bits. The owner *O*, mode *M*, size *S*, modify time *T*, and file name *F* are printed. If the file system is not mounted and the *-n* option was not specified, the entry will be removed automatically if the inode it points to is character size 0.

Possible responses to REMOVE prompt are:

YES                      The directory entry *F* is removed.

NO                        Ignore this error condition.

**DUP/BAD I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F (REMOVE)**

Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with directory entry *F*, directory inode *I*. The owner *O*, mode *M*, size *S*, modify time *T*, and directory name *F* are printed.

Possible responses to REMOVE prompt are:

YES                      The directory entry *F* is removed.

NO                        Ignore this error condition.

**DUP/BAD I=I OWNER=O MODE=M SIZE=S MTIME=T FILE=F (REMOVE)**

Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with directory entry *F*, inode *I*. The owner *O*, mode *M*, size *S*, modify time *T*, and file name *F* are printed.



Possible responses to REMOVE prompt are:

- YES                    The directory entry *F* is removed.
- NO                     Ignore this error condition.

**BAD BLK B IN DIR I=I OWNER=O MODE=M SIZE=S MTIME=T**

A bad block was found in DIR inode *I*. This message only occurs when the *-q* option is used. Error conditions looked for in directory blocks are nonzero padded entries, inconsistent "." and ".." entries, and imbedded slashes in the name field. This error message indicates that the user should at a later time either remove the directory inode if the entire block looks bad or change (or remove) those directory entries that look bad.

#### F. PHASE 3: CHECK CONNECTIVITY

This phase concerns itself with the directory connectivity seen in Phase 2. This part lists error conditions resulting from unreferenced directories and missing or full *lost+found* directories.

**UNREF DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (RECONNECT)**

The directory inode *I* was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of directory inode *I* are printed. *Fsck* will force the reconnection of a nonempty directory.

Possible responses to RECONNECT prompt are:

- YES                    Reconnect directory inode *I* to the file system in directory for lost files (usually *lost+found*). This may invoke *lost+found* error condition in Phase 3 if there are problems connecting directory inode *I* to *lost+found*. This may also invoke CONNECTED error condition in Phase 3 if link was successful.
- NO                     Ignore this error condition. This will always invoke UNREF error condition in Phase 4.

**SORRY. NO *lost+found* DIRECTORY**

There is no *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a directory in *lost+found*. This will always invoke the UNREF error condition in Phase 4. Check access modes of *lost+found*. See *fsck(1M)* in the UNIX System Administrator's Manual for further details.

**SORRY. NO SPACE IN *lost+found* DIRECTORY**

There is no space to add another entry to the *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a directory in *lost+found*. This will always invoke the UNREF error condition in Phase 4. Clean out unnecessary entries in *lost+found* or make *lost+found* larger. See *fsck(1M)* in the UNIX System Administrator's Manual for further details.

**DIR I=I1 CONNECTED. PARENT WAS I=I2**

This is an advisory message indicating a directory inode *I1* was successfully connected to the *lost+found* directory. The parent inode *I2* of the directory inode *I1* is replaced by the inode number of the *lost+found* directory. See the parts "Link Count" and "Data Blocks" for more information.

#### G. PHASE 4: CHECK REFERENCE COUNTS

This phase concerns itself with the link count information seen in Phase 2 and Phase 3. This part lists error conditions resulting from unreferenced files, missing or full *lost+found* directory, incorrect link counts for files,



directories, or special files, unreferenced files and directories, bad and duplicate blocks in files and directories, and incorrect total free-inode counts.

**UNREF FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (RECONNECT)**

Inode *I* was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. (See the part "Link Count".) If the *-n* option is not set and the file system is not mounted, empty files will not be reconnected and will be cleared automatically.

Possible responses to RECONNECT prompt are:

**YES** Reconnect inode *I* to file system in the directory for lost files (usually *lost+found*). This may invoke *lost+found* error condition in Phase 4 if there are problems connecting inode *I* to *lost+found*.

**NO** Ignore this error condition. This will always invoke CLEAR error condition in Phase 4.

**SORRY. NO *lost+found* DIRECTORY**

There is no *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a file in *lost+found*. This will always invoke CLEAR error condition in Phase 4. Check access modes of *lost+found*.

**SORRY. NO SPACE IN *lost+found* DIRECTORY**

There is no space to add another entry to the *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a file in *lost+found*. This will always invoke the CLEAR error condition in Phase 4. Check size and contents of *lost+found*.

**(CLEAR)**

The inode mentioned in the immediately previous error condition can not be reconnected. See the part "Link Count" for more information.

Possible responses to CLEAR prompt are:

**YES** Deallocate inode mentioned in the immediately previous error condition by zeroing its contents.

**NO** Ignore this error condition.

**LINK COUNT FILE I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST)**

The link count for inode *I*, which is a file, is *X* but should be *Y*. The owner *O*, mode *M*, size *S*, and modify time *T* are printed. See the part "Link Count" for more information.

Possible responses to ADJUST prompt are:

**YES** Replace link count of file inode *I* with *Y*.

**NO** Ignore this error condition.

**LINK COUNT DIR I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST)**

The link count for inode *I*, which is a directory, is *X* but should be *Y*. The owner *O*, mode *M*, size *S*, and modify time *T* of directory inode *I* are printed.



Possible responses to ADJUST prompt are:

- YES                    Replace link count of directory inode *I* with *Y*.
- NO                    Ignore this error condition.

**LINK COUNT *F* *I*=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* COUNT=*X* SHOULD BE *Y* (ADJUST)**

The link count for *F* inode *I* is *X* but should be *Y*. The file name *F*, owner *O*, mode *M*, size *S*, and modify time *T* are printed.

Possible responses to ADJUST prompt are:

- YES                    Replace link count of inode *I* with *Y*.
- NO                    Ignore this error condition.

**UNREF FILE *I*=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (CLEAR)**

Inode *I*, which is a file, was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. (See the parts "Link Counts" and "Data Blocks".) If the *-n* option is not set and the file system is not mounted, empty files will be cleared automatically.

Possible responses to CLEAR prompt are:

- YES                    Deallocate inode *I* by zeroing its contents.
- NO                    Ignore this error condition.

**UNREF DIR *I*=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (CLEAR)**

Inode *I*, which is a directory, was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. If the *-n* option is not set and the file system is not mounted, empty directories will be cleared automatically. Nonempty directories will not be cleared.

Possible responses to CLEAR prompt are:

- YES                    Deallocate inode *I* by zeroing its contents.
- NO                    Ignore this error condition.

**BAD/DUP FILE *I*=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (CLEAR)**

Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with file inode *I*. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. See the parts "Duplicate Blocks" and "Bad Blocks" for more information.

Possible responses to CLEAR prompt are:

- YES                    Deallocate inode *I* by zeroing its contents.
- NO                    Ignore this error condition.



**BAD/DUP DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR)**

Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with directory inode *I*. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed.

Possible responses to CLEAR prompt are:

- YES                      Deallocate inode *I* by zeroing its contents.  
NO                        Ignore this error condition.

**FREE INODE COUNT WRONG IN SUPERBLK (FIX)**

The actual count of the free inodes does not match the count in the superblock of the file system. (See the part "Free-Inode Count".) If the **-q** option is specified, the count will be fixed automatically in the superblock.

Possible responses to FIX prompt are:

- YES                      Replace count in superblock by actual count.  
NO                        Ignore this error condition.

**H. PHASE 5: CHECK FREE LIST**

This phase concerns itself with the free-block list. This part lists error conditions resulting from bad blocks in the free-block list, bad free-blocks count, duplicate blocks in the free-block list, unused blocks from the file system not in the free-block list, and the total free-block count incorrect.

**EXCESSIVE BAD BLKS IN FREE LIST (CONTINUE)**

The free-block list contains more than a tolerable number (usually 10) of blocks with a value less than the first data block in the file system or greater than the last block in the file system. See the parts "Free-Block List" and "Bad Blocks" for more information.

Possible responses to CONTINUE prompt are:

- YES                      Ignore rest of the free-block list and continue execution of **fsck**. This error condition will always invoke "BAD BLKS IN FREE LIST" error condition in Phase 5.  
NO                        Terminate program.

**EXCESSIVE DUP BLKS IN FREE LIST (CONTINUE)**

The free-block list contains more than a tolerable number (usually 10) of blocks claimed by inodes or earlier parts of the free-block list.

Possible responses to CONTINUE prompt are:

- YES                      Ignore the rest of the free-block list and continue execution of **fsck**. This error condition will always invoke "DUP BLKS IN FREE LIST" error condition in Phase 5.  
NO                        Terminate program.



**BAD FREEBLK COUNT**

The count of free blocks in a free-list block is greater than 50 or less than 0. This error condition will always invoke the "BAD FREE LIST" condition in Phase 5.

**X BAD BLKS IN FREE LIST**

X blocks in the free-block list have a block number lower than the first data block in the file system or greater than the last block in the file system. This error condition will always invoke the "BAD FREE LIST" condition in Phase 5. See the parts "Free-Block List" and "Bad Blocks" for more information.

**X DUP BLKS IN FREE LIST**

X blocks claimed by inodes or earlier parts of the free-list block were found in the free-block list. This error condition will always invoke the "BAD FREE LIST" condition in Phase 5.

**X BLK(S) MISSING**

X blocks unused by the file system were not found in the free-block list. This error condition will always invoke the "BAD FREE LIST" condition in Phase 5. See the part "Free-Block List" for more information.

**FREE BLK COUNT WRONG IN SUPERBLOCK (FIX)**

The actual count of free blocks does not match the count in the superblock of the file system. See the part "Free-Block Count" for more information.

Possible responses to FIX prompt are:

YES                    Replace count in superblock by actual count.

NO                    Ignore this error condition.

**BAD FREE LIST (SALVAGE)**

Phase 5 has found bad blocks in the free-block list, duplicate blocks in the free-block list, or blocks missing from the file system. If the -q option is specified, the free-block list will be salvaged automatically.

Possible responses to SALVAGE prompt are:

YES                    Replace actual free-block list with a new free-block list. The new free-block list will be ordered to reduce time spent by the disk waiting for the disk to rotate into position.

NO                    Ignore this error condition.

**I. PHASE 6: SALVAGE FREE LIST**

This phase concerns itself with the free-block list reconstruction. This part lists error conditions resulting from the blocks-to-skip and blocks-per-cylinder values.

**Default free-block list spacing assumed**

This is an advisory message indicating the blocks-to-skip is greater than the blocks-per-cylinder, the blocks-to-skip is less than one, the blocks-per-cylinder is less than one, or the blocks-per-cylinder is greater than 1000.



The default values of 7 blocks-to-skip and 400 blocks-per-cylinder are used. See **fsck(1M)** in the UNIX System Administrator's Manual for further details.

#### J. CLEANUP

Once a file system has been checked, a few cleanup functions are performed. This part lists advisory messages about the file system and modify status of the file system.

##### **X files Y blocks Z free**

This is an advisory message indicating that the file system checked contained *X* files using *Y* blocks leaving *Z* blocks free in the file system.

##### **\*\*\*\*\* BOOT UNIX (NO SYNC!) \*\*\*\*\***

This is an advisory message indicating that a mounted file system or the root file system has been modified by **fsck**. If the UNIX system is not rebooted immediately, the work done by **fsck** may be undone by the in-core copies of tables the UNIX system keeps.

##### **\*\*\*\*\* FILE SYSTEM WAS MODIFIED \*\*\*\*\***

This is an advisory message indicating that the current file system was modified by **fsck**. If this file system is mounted or is the current root file system, **fsck** should be halted and the UNIX system rebooted. If the UNIX system is not rebooted immediately, the work done by **fsck** may be undone by the in-core copies of tables.



NOTES



all, including the following information: (1) the name of the person or persons who prepared the report; (2) the date when the report was prepared; (3) the name of the person or persons to whom the report was submitted; (4) the name of the person or persons who reviewed the report; (5) the name of the person or persons who approved the report; (6) the name of the person or persons who distributed the report; (7) the name of the person or persons who received the report; (8) the name of the person or persons who filed the report; (9) the name of the person or persons who maintained the report; (10) the name of the person or persons who disposed of the report.

## LP Spooling System

1. The LP Spooling System is a system for spooling data from a magnetic tape to a paper tape. It consists of a magnetic tape reader, a spooling unit, and a paper tape writer. The magnetic tape reader reads data from a magnetic tape and sends it to the spooling unit. The spooling unit then sends the data to the paper tape writer, which writes the data on a paper tape.

2. The LP Spooling System is a system for spooling data from a magnetic tape to a paper tape. It consists of a magnetic tape reader, a spooling unit, and a paper tape writer. The magnetic tape reader reads data from a magnetic tape and sends it to the spooling unit. The spooling unit then sends the data to the paper tape writer, which writes the data on a paper tape.

3. The LP Spooling System is a system for spooling data from a magnetic tape to a paper tape. It consists of a magnetic tape reader, a spooling unit, and a paper tape writer. The magnetic tape reader reads data from a magnetic tape and sends it to the spooling unit. The spooling unit then sends the data to the paper tape writer, which writes the data on a paper tape.



This document is part of the ADMINISTRATOR'S GUIDE. Therefore the pagenumbers don't begin with 1.

**Trademarks:**

|               |                       |
|---------------|-----------------------|
| MUNIX, CADMUS | for PCS               |
| DEC, PDP      | for DEC               |
| UNIX          | for Bell Laboratories |

Copyright 1984 by  
PCS GmbH, Pfälzer-Wald-Strasse 36, D-8000 München 90, tel. (089) 67804-0

The information contained herein is the property of PCS and shall neither be reproduced in whole or in part without PCS's prior written approval nor be implied to grant any license to make, use or sell equipment manufactured herewith.

PCS reserves the right to make changes without notice in the specifications and materials contained herein and shall not be responsible for any damages (including consequential) caused by reliance on the materials presented.



## 9. LP SPOOLING SYSTEM

### GENERAL

The LP program is a system of commands which performs diverse spooling functions under the UNIX operating system. Since the primary LP application is off-line printing, this document focuses mainly on spooling to line printers. LP allows administrators to customize the system to spool to a collection of line printers of any type and to group printers into logical classes in order to maximize the throughput of the devices. Users are provided the capabilities of queuing and canceling print requests, preventing and allowing queuing to and printing on devices, starting and stopping LP from processing requests, changing configuration of printers and finding status of the LP system. This section describes the role of an LP Administrator in performing restricted functions and overseeing the smooth operation of LP.

Throughout this section, each reference of the form **name(1M)**, **name(7)**, or **name(8)** refers to entries in the UNIX System Administrator's Manual. All other references to entries of the form **name(N)**, where "N" is a number (1 through 6) possibly followed by a letter, refers to entry **name** in Section "N" of the UNIX System User's Manual.

### OVERVIEW OF LP FEATURES

#### A. Definitions

Several terms must be defined before presenting a brief summary of LP commands. The LP was designed with the flexibility to meet the needs of users on different UNIX systems. Changes to the LP configuration are performed by the **lpadmin(1M)** command.

LP makes a distinction between printers and printing devices. A *device* is a physical peripheral device or a file and is represented by a full UNIX system pathname. A *printer* is a logical name that represents a device. At different points in time, a printer may be associated with different devices. A *class* is a name given to an ordered list of printers. Every class must contain at least one printer. Each printer may be a member of zero or more classes. A *destination* is a printer or a class. One destination may be designated as the *system default destination*. The **lp(1)** command will direct all output to this destination unless the user specifies otherwise. Output that is routed to a printer will be printed only by that printer, whereas output directed to a class will be printed by the first available class member.

Each invocation of **lp** creates an output request that consists of the files to be printed and options from the **lp** command line. An interface program which formats requests must be supplied for each printer. The LP scheduler, **lp sched(1M)**, services requests for all destinations by routing requests to interface programs to do the printing on devices. An LP configuration for a system consists of devices, destinations, and interface programs.

#### B. Commands

##### Commands for General Use

The **lp(1)** command is used to request the printing of files. It creates an output request and returns a request id of the form

**dest-seqno**

to the user, where *seqno* is a unique sequence number across the entire LP system, and *dest* is the destination where the request was routed.

**Cancel** is used to cancel output requests. The user supplies request ids as returned by **lp** or printer names, in which case the currently printing requests on those printers are canceled.



**Disable** prevents **lpsched** from routing output requests to printers.

**Enable(1)** allows **lpsched** to route output requests to printers.

#### Commands for LP Administrators

Each LP system must designate a person or persons as LP administrator to perform the restricted functions listed below. Either the superuser or any user who is logged into the UNIX system as **lp** qualifies as an LP Administrator. All LP files and commands are owned by **lp**, except for **lpadmin** and **lpsched** which are owned by root. The following commands will be described in more detail later in this section.

- Lpadmin(1M)**      Modifies LP configuration. Many features of this command cannot be used when **lpsched** is running.
- Lpsched(1M)**      Routes output requests to interface programs which do the printing on devices.
- Lpshut**            Stops **lpsched** from running. All printing activity is halted, but other LP commands may still be used.
- Accept(1M)**        Allows **lp** to accept output requests for destinations.
- Reject**            Prevents **lp** from accepting requests for destinations.
- Lpmove**            Moves output requests from one destination to another. Whole destinations may be moved at once. This command cannot be used when **lpsched** is running.

#### BUILDING LP

All LP commands are built from source code that resides in the `/usr/src/cmd/lp` directory including the make file, *lp.mk*. Unless some of the definitions in *lp.mk* are changed, LP may be installed only by the superuser. Before installing a new LP system, make sure there is a login called **lp** on your system and that the spool directory, `/usr/spool/lp`, does not exist. To install LP, perform the following:

```
cd /usr/src/cmd/lp
make -f lp.mk install
```

This builds all LP commands and creates an initial LP configuration consisting of no printers, classes, or default destination. LP must be configured by an LP administrator using the **lpadmin** command in order to create a useful spooler.

In addition, add the following code to `/etc/rc`:

```
rm -f /usr/spool/lp/SCHEDLOCK
/usr/lib/lpsched
echo " LP scheduler started "
```

This starts the LP scheduler each time that the UNIX system is restarted.

Several variables in *lp.mk* may be changed before installing LP to customize the system:

| Variable | Default Value              | Meaning                     |
|----------|----------------------------|-----------------------------|
| SPOOL    | <code>/usr/spool/lp</code> | spool directory             |
| ADMIN    | <code>lp</code>            | logname of LP Administrator |



|        |                 |                               |
|--------|-----------------|-------------------------------|
| GROUP  | <i>bin</i>      | group owning LP commands/data |
| ADMDIR | <i>/usr/lib</i> | commands of administrator     |
| USRDIR | <i>/usr/bin</i> | user commands reside here     |

If an existing LP spool directory is corrupted (but not the LP programs) or if it needs to be rebuilt from scratch, make sure that **lpsched** is not running and perform the following as superuser:

1. Make copies of any interface programs that are not standard LP software. **DO NOT** make these copies underneath the spool directory. The pathname for printer "p" is */usr/spool/lp/interface/p*.
2. `rm -fr /usr/spool/lp`
3. Make `-f lp.mk new`. (This recreates the bare LP configuration described above.)

#### PRECAUTIONS:

1. Some LP commands invoke other LP commands. Moving them after they are built will cause some commands to fail.
2. The files under the SPOOL directory should be modified **only by LP commands**.
3. All LP commands require set-user-id permission. If this is removed, the commands will fail.

#### CONFIGURING LP—THE "lpadmin" COMMAND

Changes to the LP configuration should be made by using the **lpadmin** command and not by hand. **Lpadmin** will not attempt to alter the LP configuration when **lpsched** is running, except where explicitly noted below.

##### A. Introducing New Destinations

The following information must be supplied to **lpadmin** when introducing a new printer:

1. The printer name (`-p printer`) is an arbitrary name which must conform to the following rules:
  - It must be no longer than 14 characters.
  - It must consist solely of alphanumeric characters and underscores.
  - It must not be the name of an existing LP destination (printer or class).
2. The device associated with the printer (`-v device`). This is the pathname of a hardwired printer, a login terminal, or other file that is writable by lp.
3. The printer interface program. This may be specified in one of three ways:
  - It may be selected from a list of model interfaces supplied with LP (`-m model`).
  - It may be the same interface that an existing printer uses (`-e printer`).
  - It may be a program supplied by the LP administrator (`-i interface`).

Information which need not always be supplied when creating a new printer includes:

1. The user may specify `-h` to indicate that the device for the printer is hardwired or the device is the name of a file (this is assumed by default). If, on the other hand, the device is the pathname of a login terminal,



then `-l` must be included on the command line. This indicates to *lpsched* that it must automatically disable this printer each time *lpsched* starts running. This fact is reported by *lpstat* when it indicates printer status:

```
$ lpstat -pa
printer a (login terminal) disabled Oct 31 11:15—
disabled by scheduler: login terminal
```

This is done because device names for login terminals can be (and usually are) associated with different physical devices from day to day. If the scheduler did not take this action, somebody might log in and be surprised that LP is spooling to his/her terminal!

2. The new printer may be added to an existing class or added to a new class (`-cclass`). New class names must conform to the same rules for new printer names.

## EXAMPLES

The following examples will be referenced by further examples in later sections.

1. Create a printer called `pr1` whose device is `/dev/printer` and whose interface program is the model `hp` interface:

```
$ /usr/lib/lpadmin -ppr1 -v/dev/printer -mhp
```

2. Add a printer called `pr2` whose device is `/dev/tty22` and whose interface is a variation of the model `prx` interface. It is also a login terminal:

```
$ cp /usr/spool/lp/model/prx xxx
< edit xxx >
$ /usr/lib/lpadmin -ppr2 -v/dev/tty22 -ixxx -l
```

3. Create a printer called `pr3` whose device is `/dev/tty23`. The `pr3` will be added to a new class called `cl1` and will use the same interface as printer `pr2`:

```
$ /usr/lib/lpadmin -ppr3 -v/dev/tty23 -epr2 -ccl1
```

## B. Modifying Existing Destinations

Modifications to existing destinations must always be made with respect to a printer name (`-p printer`). The modifications may be one or more of the following:

1. The device for the printer may be changed (`-v device`). If this is the only modification, then this may be done even while *lpsched* is running. This facilitates changing devices for login terminals.
2. The printer interface program may be changed (`-m model`, `-e printer`, `-i interface`).
3. The printer may be specified as hardwired (`-h`) or as a login terminal (`-l`).
4. The printer may be added to a new or existing class (`-cclass`).
5. The printer may be removed from an existing class (`-r class`). Removing the last remaining member of a class causes the class to be deleted. No destination may be removed if it has pending requests. In that case, `lpmove` or `cancel` should be used to move or delete the pending requests.



**EXAMPLES**

These examples are based on the LP configuration created by those in the previous section.

1. Add printer pr2 to class cl1:

```
$ /usr/lib/lpadmin -ppr2 -ccl1
```

2. Change pr2's interface program to the model prx interface, change its device to /dev/tty24, and add it to a new class called cl2:

```
$ /usr/lib/lpadmin -ppr2 -mprx -v/dev/tty24 -ccl2
```

Note that printers pr2 and pr3 now use different interface programs even though pr3 was originally created with the same interface as pr2. Printer pr2 is now a member of two classes.

3. Specify printer pr2 as a hardwired printer:

```
$ /usr/lib/lpadmin -ppr2 -h
```

4. Add printer pr1 to class cl2:

```
$ /usr/lib/lpadmin -ppr1 -ccl2
```

The members of class cl2 are now pr2 and pr1, in that order. Requests routed to class cl2 will be serviced by pr2 if both pr2 and pr1 are ready to print; otherwise, they will be printed by the one which is next ready to print.

5. Remove printers pr2 and pr3 from class cl1:

```
$ /usr/lib/lpadmin -ppr2 -rccl1
```

```
$ /usr/lib/lpadmin -ppr3 -rccl1
```

Since pr3 was the last remaining member of class cl1, the class is removed.

6. Add pr3 to a new class called cl3.

```
$ /usr/lib/lpadmin -ppr3 -ccl3
```

**C. Specifying the System Default Destination**

The system default destination may be changed even when **lpsched** is running.

**EXAMPLES**

1. Establish class cl1 as the system default destination:

```
$ /usr/lib/lpadmin -dcl1
```

2. Establish no default destination:

```
$ /usr/lib/lpadmin -d
```

**D. Removing Destinations**

Classes and printers may be removed only if there are no pending requests that were routed to them. Pending requests must either be canceled using **cancel** or moved to other destinations using **lpmove** before destinations may be removed. If the removed destination is the system default destination, then the system will have



no default destination until the default destination is respecified. When the last remaining member of a class is removed, then the class is also removed. The removal of a class never implies the removal of printers.

### EXAMPLES

1. Make printer pr1 the system default destination:

```
$ /usr/lib/lpadmin -dpr1
```

Remove printer pr1:

```
$ /usr/lib/lpadmin -xpr1
```

Now there is no system default destination.

2. Remove printer pr2:

```
$ /usr/lib/lpadmin -xpr2
```

Class cl2 is also removed since pr2 was its only member.

3. Remove class cl3:

```
$ /usr/lib/lpadmin -xcl3
```

Class cl3 is removed, but printer pr3 remains.

### MAKING AN OUTPUT REQUEST—THE "lp" COMMAND

Once LP destinations have been created, users may request output by using the **lp** command. The request id that is returned may be used to see if the request has been printed or to cancel the request.

The LP program determines the destination of a request by checking the following list in order:

- If the user specifies **-d dest** on the command line, then the request is routed to *dest*.
- If the environment variable **LPDEST** is set, the request is routed to the value of **LPDEST**.
- If there is a system default destination, then the request is routed there.
- Otherwise, the request is rejected.

### EXAMPLES

1. There are at least four ways to print the password file on the system default destination:

```
lp /etc/passwd  
lp < /etc/passwd  
cat /etc/passwd | lp  
lp -c /etc/passwd
```

The last three ways cause copies of the file to be printed, whereas the first way prints the file directly. Thus, if the file is modified between the time the request is made and the time it is actually printed, then the changes will be reflected in the output.



2. Print two copies of file abc on printer xyz and title the output "my file":

```
pr abc | lp -dxyz -n2 -t "my file"
```

3. Print file xxx on a Diablo\* 1640 printer called zoo in 12-pitch and write to the user's terminal when printing has completed:

```
lp -dzoo -o12 -w xxx
```

In this example, "12" is an option that is meaningful to the model Diablo 1640 interface program that prints output in 12-pitch mode [see `lpadmin(1M)`].

#### FINDING LP STATUS—LPSTAT

The `lpstat` command is used to find status information about LP requests, destinations, and the scheduler.

#### EXAMPLES

1. List the status of all pending output requests made by this user:

```
lpstat
```

The status information for a request includes the request id, the logname of the user, the total number of characters to be printed, and the date and time the request was made.

2. List the status of printers p1 and p2:

```
lpstat -pp1,p2
```

#### CANCELING REQUESTS—CANCEL

The LP requests may be canceled using the `cancel` command. Two kinds of arguments may be given to the command—request ids and printer names. The requests named by the request ids are canceled and requests that are currently printing on the named printers are canceled. Both types of arguments may be intermixed.

#### EXAMPLE

Cancel the request that is now printing on printer xyz:

```
cancel xyz
```

If the user that is canceling a request is not the same one that made the request, then mail is sent to the owner of the request. LP allows any user to cancel requests in order to eliminate the need for users to find LP administrators when unusual output should be purged from printers.

#### ALLOWING AND REFUSING REQUESTS—ACCEPT AND REJECT

When a new destination is created, `lp` will reject requests that are routed to it. When the LP administrator is sure that it is set up correctly, he or she should allow `lp` to accept requests for that destination. The `accept` command performs this function.

Sometimes it is necessary to prevent `lp` from routing requests to destinations. If printers have been removed or are waiting to be repaired or if too many requests are building for printers, then it may be desirable to cause

\* Trademark of Diablo Systems, Inc.



**lp** to reject requests for those destinations. The **reject** command performs this function. After the condition that led to the rejection of requests has been remedied, the **accept** command should be used to allow requests to be taken again.

The acceptance status of destinations is reported by the **-a** option of **lpstat**.

### EXAMPLES

1. Cause **lp** to reject requests for destination **xyz**:

```
/usr/lib/reject -r "printer xyz needs repair" xyz
```

Any users that try to route requests to **xyz** will encounter the following:

```
$ lp -dxyz file
lp: can not accept requests for destination "xyz "
    -printer xyz needs repair
```

2. Allow **lp** to accept requests routed to destination **xyz**:

```
/usr/lib/accept xyz
```

### ALLOWING AND INHIBITING PRINTING—ENABLE AND DISABLE

The **enable** command allows the LP scheduler to print requests on printers. That is, the scheduler routes requests only to the interface programs of enabled printers. Note that it is possible to enable a printer but to prevent further requests from being routed to it.

The **disable** command cancels the effects of the **enable** command. It prevents the scheduler from routing requests to printers, independently of whether or not **lp** is allowing them to accept requests. Printers may be disabled for several reasons including malfunctioning hardware, paper jams, and end of day shutdowns. If a printer is busy at the time it is disabled, then the request that it was printing will be reprinted in its entirety either on another printer (if the request was originally routed to a class of printers) or on the same one when the printer is reenabled. The **-c** option causes the currently printing requests on busy printers to be canceled in addition to disabling the printers. This is useful if strange output is causing a printer to behave abnormally.

### EXAMPLE

Disable printer **xyz** because of a paper jam:

```
$ disable -r "paper jam" xyz
printer "xyz" now disabled
```

Find the status of printer **xyz**:

```
$ lpstat -pxyz
printer "xyz" disabled since Jan 5 10:15 —
    paper jam
```

Now, reenable **xyz**:

```
$ enable xyz
printer "xyz" now enabled
```



### MOVING REQUESTS BETWEEN DESTINATIONS—LPMOVE

Occasionally, it is useful for LP administrators to move output requests between destinations. For instance, when a printer is down for repairs, it may be desirable to move all of its pending requests to a working printer. This is one way to use the **lpmove** command. The other use of this command is to move specific requests to a different destination. **Lpmove** will refuse to move requests while the LP scheduler is running.

#### EXAMPLES

1. Move all requests for printer abc to printer xyz:

```
$ /usr/lib/lpmove abc xyz
```

All of the moved requests are renamed from abc-nnn to xyz-nnn. As a side effect, destination abc is no longer accepting further requests.

2. Move requests zoo-543 and abc-1200 to printer xyz:

```
$ /usr/lib/lpmove zoo-543 abc-1200 xyz
```

The two requests are now renamed xyz-543 and xyz-1200.

### STOPPING AND STARTING THE SCHEDULER—LPSHUT AND LPSCHED

**Lpsched** is the program that routes the output requests that were made with **lp** through the appropriate printer interface programs to be printed on line printers. Each time the scheduler routes a request to an interface program, it records an entry in the log file, `/usr/spool/lp/log`. This entry contains the logname of the user that made the request, the request id, the name of the printer that the request is being printed on, and the date and time that printing first started. In the case that a request has been restarted, more than one entry in the log file may refer to the request. The scheduler also records error messages in the log file. When **lpsched** is started, it renames `/usr/spool/lp/log` to `/usr/spool/lp/oldlog` and starts a new log file.

No printing will be performed by the LP system unless **lpsched** is running. Use the command

```
lpstat -r
```

to find the status of the LP scheduler.

**Lpsched** is normally started by the `/etc/rc` program as described above and continues to run until the UNIX system is shut down. The scheduler operates in the `/usr/spool/lp` directory. When it starts running, it will exit immediately if a file called **SCHEDLOCK** exists. Otherwise, it creates this file in order to prevent more than one scheduler from running at the same time.

Occasionally, it is necessary to shut down the scheduler in order to reconfigure LP or to rebuild the LP software. The command

```
/usr/lib/lpshut
```

causes **lpsched** to stop running and terminates all printing activity. All requests that were in the middle of printing will be reprinted in their entirety when the scheduler is restarted.

To restart the LP scheduler, use the command

```
/usr/lib/lpsched
```



Shortly after this command is entered, **lpstat** should report that the scheduler is running. If not, it is possible that a previous invocation of **lpsched** exited without removing **SCHEDLOCK**, so try the following:

```
rm -f /usr/spool/lp/SCHEDLOCK
    /usr/lib/lpsched
```

The scheduler should be running now.

#### PRINTER INTERFACE PROGRAMS

Every LP printer must have an interface program which does the actual printing on the device that is currently associated with the printer. Interface programs may be shell procedures, C programs, or any other executable programs. The LP model interfaces are all written as shell procedures and can be found in the */usr/spool/lp/model* directory. At the time **lpsched** routes an output request to a printer P, the interface program for P is invoked in the directory */usr/spool/lp* as follows:

```
interface/P id user title copies options file ...
where
id is the request id returned by lp
user is logname of user who made the request
title is optional title specified by the user
copies is number of copies requested by user
options is a blank-separated list of class or
printer-dependent options specified by user
file is the full pathname of a file to be printed
```

#### EXAMPLES

The following examples are requests made by user "smith" with a system default destination of printer "xyz". Each example lists an **lp** command line followed by the corresponding command line generated for printer xyz's interface program:

1. **lp /etc/passwd /etc/group**  
     interface/xyz xyz-52 smith " " 1 " " /etc/passwd /etc/group
2. **pr /etc/passwd | lp -t " users " -n5**  
     interface/xyz xyz-53 smith users 5 " "  
     /usr/spool/lp/request/xyz/d0-53
3. **lp /etc/passwd -oa -ob**  
     interface/xyz xyz-54 smith " " 1 " a b " /etc/passwd

When the interface program is invoked, its standard input comes from */dev/null* and both the standard output and standard error output are directed to the printer's device. Devices are opened for reading as well as



writing when file modes permit. In the case where a device is a regular file, all output is appended to the end of the file.

Given the command line arguments and the output directed to a device, interface programs may format their output in any way they choose. Interface programs must ensure that the proper stty modes (terminal characteristics such as baud rate, output options, etc.) are in effect on the output device. This may be done as follows in a shell interface only if the device is opened for reading:

```
stty mode ... <&1
```

That is, take the standard input for the **stty** command from the device.

When printing has completed, it is the responsibility of the interface program to exit with a code indicative of the success of the print job. Exit codes are interpreted by **lpsched** as follows:

| CODE             | MEANING TO LPSCHED                                                                                                                                                                                                                             |
|------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| zero             | The print job has completed successfully.                                                                                                                                                                                                      |
| 1 to 127         | A problem was encountered in printing this particular request (e.g., too many nonprintable characters). This problem will not affect future print jobs. <b>Lpsched</b> notifies users by mail that there was an error in printing the request. |
| greater than 127 | These codes are reserved for internal use by <b>lpsched</b> . Interface programs must not exit with codes in this range.                                                                                                                       |

When problems that are likely to affect future print jobs occur (e.g., a device filter program is missing), the interface programs would be wise to disable printers so that print requests are not lost. When a busy printer is disabled, the interface program will be terminated with signal 15.

#### SETTING UP HARDWIRED DEVICES AND LOGIN TERMINALS AS LP PRINTERS

##### A. Hardwired Devices

As an example of how to set up a hardwired device for use as an LP printer, let us consider using tty line 15 as printer xyz. As superuser, perform the following:

1. Avoid unwanted output from non-LP processes and ensure that LP can write to the device:

```
$ chown lp /dev/tty15
$ chmod 600 /dev/tty15
```

2. Change **/etc/inittab** so that tty15 is not a login terminal. In other words, ensure that **/etc/getty** is not trying to log users in at this terminal. Change the entries for line 15 to:

```
1:15:o:
2:15:o:
```

Enter the command:

```
$ init 2
```

If there is currently an invocation of **/etc/getty** running on tty15, kill it. Now, and when the UNIX system is rebooted, tty15 will be initialized with default stty modes. Thus, it is up to LP interface programs to establish the proper baud rate and other stty modes for correct printing to occur.



3. Introduce printer xyz to LP using the model prx interface program:

```
$ /usr/lib/lpadmin -pxyz -v/dev/tty15 -mprx
```

4. When xyz is created, it will initially be disabled and lp will be rejecting requests routed to it. If it is desired, allow lp to accept requests for xyz:

```
/usr/lib/accept xyz
```

This will allow requests to build up for xyz, and to be printed when it is enabled at a later time.

5. When it is desired for printing to occur, be sure that the printer is ready to receive output. For several printers, this means that the top of form has been adjusted and that the printer is on-line. Enable printing to occur on xyz:

```
enable xyz
```

When requests have been routed to xyz, they will begin printing.

#### B. Login Terminals

Login terminals may also be used as LP printers. To do this for a Diablo 1640 terminal called abc, perform the following:

1. Introduce printer abc to LP using the model 1640 interface program:

```
$ /usr/lib/lpadmin -pabc -v/dev/null -m1640 -l
```

Note that `/dev/null` is used as abc's device because we will specify the actual device each time that abc is enabled. This device may be different from day to day. When abc is created, it will initially be disabled; and lp will be rejecting requests routed to it. If it is desired, allow lp to accept requests for abc:

```
/usr/lib/accept abc
```

This will allow requests to build up for abc and to be printed when it is enabled at a later time. It is not advisable to enable abc for printing, however, until the following steps have been taken.

2. Log terminal in if this has not already been done.
3. Assuming the `tty(1)` command reports that this terminal is `/dev/tty02`, associate this device with printer abc:

```
$ /usr/lib/lpadmin -pabc -v/dev/tty02
```

Note that `lpadmin` may be used only by an LPA. If it is desired for other users to routinely perform this step, then an LPA may establish a program owned by lp or by root with set-user-id permission that performs this function.

4. When it is desired for printing to occur, be sure that the printer is ready to receive output. For several printers, this means that the top of form has been adjusted. Enable printing to occur on abc:

```
enable abc
```

When requests have been routed to abc, they will begin printing.



5. When all printing has stopped on abc or when you want it back as a regular login terminal, you may prevent it from printing more output:

```
$ disable abc
printer "abc" now disabled
```

If abc is enabled when the UNIX system is rebooted or when `lpsched` is restarted, it will be disabled automatically.

#### SUMMARY

The administrative functions of the LP administrator have been described in detail. These functions include configuring and reconfiguring LP; maintaining printer interface programs; accepting, rejecting, and moving print requests; stopping and starting the LP scheduler; and enabling and disabling printers. LP offers administrators the following advantages over other centrally supported printer packages:

- Printers may be grouped into classes.
- LP may be configured to meet the needs of each site.
- Administrators may supply interface programs to format output in any way desirable.
- LP functions are performed by simple commands and not by hand.



**NOTES**



# **UNIX System**

## **Remote Job Entry**



This document is part of the ADMINISTRATOR'S GUIDE. Therefore the pagenumbers don't begin with 1.

Trademarks:

MUNIX, CADMUS  
DEC, PDP  
UNIX

for PCS  
for DEC  
for Bell Laboratories

Copyright 1984 by  
PCS GmbH, Pfälzer-Wald-Strasse 36, D-8000 München 90, tel. (089) 67804-0

The information contained herein is the property of PCS and shall neither be reproduced in whole or in part without PCS's prior written approval nor be implied to grant any license to make, use or sell equipment manufactured herewith.

PCS reserves the right to make changes without notice in the specifications and materials contained herein and shall not be responsible for any damages (including consequential) caused by reliance on the materials presented.



## 10. UNIX SYSTEM REMOTE JOB ENTRY

### GENERAL

#### A. Purpose

This section contains information on the design and operation of the UNIX System Remote Job Entry (RJE). In this document, RJE refers to the facilities provided by UNIX operating system and *not* to the Remote Job Entry feature of the HASP and JES2 subsystems produced by International Business Machines (IBM).

The information contained in this section should be used to augment the information contained in the UNIX System Administrator's Manual [rje(8)] and the "Remote Job Entry" section of the UNIX System User's Guide. There will be assumptions made concerning allocation of responsibilities between UNIX system and IBM operations, hardware configuration, etc. Although these assumptions may not fully apply to your location, they should not interfere with the intent of this document.

The major topics discussed in this document are as follows:

- **SETTING UP**—Hardware requirements and RJE generation on the IBM and UNIX systems.
- **DIRECTORY STRUCTURES**—The controlling RJE directory structure and a typical RJE subsystem directory structure.
- **RJE PROGRAMS**—Programs that make up an RJE subsystem.
- **UTILITY PROGRAMS**—Programs available for debugging or tracing.
- **RJE ACCOUNTING**—The accounting of jobs done by RJE and some methods for using this accounting data.
- **TROUBLESHOOTING**—Error recovery and procedures for identifying and fixing RJE problems.

#### B. Facilities

Discussions will focus on a hypothetical RJE connection between a UNIX system, whose nodename is *pwba*, and an IBM 370/168, referred to as *B*. We also assume that *pwba* is connected to an IBM 370/158, referred to as *C*. The UNIX operating system machine emulates an IBM System/360 remote multileaving work station.

### SETTING UP

#### A. Hardware

In the remainder of this guide, the hardware described below will be referred to as the *physical device*; and its name will be referred to as **device?**; where ? is the device number. To use RJE on a UNIX system, the following hardware is needed:

##### DEC Hardware

- **KMC11-B Microprocessor**—Used to drive the RJE line.
- **DMC11-DA or DMC11-FA line unit**—The DMC11-DA interfaces with Bell 208 and 209 synchronous modems or equivalent. Speeds of up to 19,200 bits per second can be used. The DMC11-FA interfaces with Bell 500 A LI/5 synchronous modems or equivalent. Speeds of up to 250,000 bits per second can be used.



Each KMC/DMC pair supports a single RJE connection. On the DMC11 line unit, the Cyclic Redundancy Check (CRC) switch should be set to inhibit automatic transmission of CRC bytes. The line unit should hold the line at logical zero when inactive.

### 3B20S Processor Hardware

- TN82—The user interface board equipped with firmware for synchronous communications.
- UN53—Synchronous communications board. This board contains hardware to handle up to four synchronous lines. One line can be high-speed with V.35 or EIA interface.

One TN82/UN53 pair and the RJE software currently support up to four low-speed (9.6 KB or lower) RJE lines or one high-speed (56 KB) RJE line.

### B. IBM Generation

The following applies to the host IBM system. The remote line to the UNIX operating system machine should be described as a System/360 remote work station. The following parameters must be initialized and must agree with their counterparts on the UNIX operating system machine:

- Number of printers (NUMPR)—The number of logical printers (up to 7)
- Number of punches (NUMPU)—The number of logical punches (up to 7)
- Number of readers (NUMRD)—The number of logical readers (up to 7).

The JES2 parameters for the hypothetical connection to IBM system *B* are as follows:

```
RMT5 S/360,LINE=5,CONSOLE,MULTI,TRANSP,NUMPR=5,
      NUMPU=1,NUMRD=5,ROUTECD=5
R5.PR1 PRWIDTH=132
R5.PR2 PRWIDTH=132
R5.PR3 PRWIDTH=132
R5.PR4 PRWIDTH=132
R5.PR5 PRWIDTH=132
R5.PU1 NOSUSPND
R5.RD1 PRIOINC=0,PRIOLIM=14
R5.RD2 PRIOINC=0,PRIOLIM=14
R5.RD3 PRIOINC=0,PRIOLIM=14
R5.RD4 PRIOINC=0,PRIOLIM=14
R5.RD5 PRIOINC=0,PRIOLIM=14
```

System *pwba* is referenced by line 5 (LINE=5), remote 5 (RMT5). It is defined as having a console, for the *rjstat(1C)* command, five printers, one punch, and five readers. Although you may have up to seven printers or punches, the total number of printers and punches may not exceed eight. The line is described as a transparent (TRANSP), multileaving (MULTI) line. The remaining information describes attributes of the printers, punches, and readers.

Normally, separator pages are transmitted with IBM print files. UNIX system RJE does not remove separator pages. To prevent transmission of separator pages on printer 1 of the previous example, its attributes would be:

```
R5.PR1 PRWIDTH=132,NOSEP
```

NOSEP should be included for all printers when separator pages are not desired. Most IBM systems can also be told via a console command to cancel transmission of separator pages on printers. This can be done from the



IBM system console or from the remote UNIX operating system machine via `rjestat`. For example, the following JES2 command would cancel separator page transmission on printer 1:

```
$TR5.PR1,S=N
```

### C. UNIX System Generation

If the RJE remote dialing facility is to be used, the administrator must make sure that the definition for RJEUCU in the file `/usr/include/rje.h` is the device to be used for remote dialing. RJEUCU is defined to be `/dev/dn2` when distributed. To compile and install RJE, the normal `make(1)` procedures are used (see the appropriate "Setting up the UNIX System" section of this guide). Once an RJE subsystem has been installed, the remote line must be described in the configuration file `/usr/rje/lines`. This file as it exists on the hypothetical system `pwba` is as follows:

```
B pwba /usr/rje1 rje1 vpm0 5:5:1 1200:512:y
C pwba /usr/rje2 rje2 vpm1 1:1:1 1200:512
```

The `/usr/rje/lines` is accessed by all components of RJE. Each line of the table (maximum of 8) defines an RJE connection. Its seven columns may be labeled *host*, *system*, *directory*, *prefix*, *device*, *peripherals*, and *parameters*. These columns are described as follows:

- *host*—The IBM System name, e.g., *A*, *B*, *C*. This string can be up to six characters long.
- *system*—The UNIX System nodename [see `uname(1)`].
- *directory*—The directory name of the servicing RJE subsystem (e.g., `/usr/rje2`).
- *prefix*—The string prepended to most files and programs in the *directory* (i.e., `rje2`).
- *device*—The name of the controlling Virtual Protocol Machine (VPM) device, with `/dev/` excised. In order to specify a VPM device, all VPM software must be installed, and the proper special files must be made [see `vpm(7)` and `mknod(1M)`].
- *peripherals*—Information on the logical devices (readers, printers, punches) used by RJE. There are three subfields. Each subfield is separated by ":" and is described as follows:
  1. Number of logical readers.
  2. Number of logical printers.
  3. Number of logical punches.

**Note:** The number of peripherals specified for an RJE subsystem *must* agree with the number of peripherals that have been described on the remote machine for that line.

- *parameters*—This field contains information on the type of connection to make. Each subfield is separated by ":". Any or all fields may be omitted; however, the fields are positional. All but trailing delimiters must be present. For example, in:

```
1200:512:::9-555-1212
```

subfields 3 and 4 are missing. Each subfield is defined as follows:

1. *space*—This subfield specifies the amount of space (*S*) in blocks that RJE tries to maintain on file systems it touches. The default is 0 blocks. `Send(1C)` will not submit jobs and `rjeinit` issues



a warning when less than 1.5S blocks are available; *rjerecv* stops accepting output from the host when the capacity falls to S blocks; RJE becomes dormant until conditions improve. If the space on the file system specified by the user on the "usr=" card would be depleted to a point below S, the file will be put in the *job* subdirectory of the connection's home directory rather than in the place that the user requested.

2. *size*—This subfield specifies the size in blocks of the largest file that can be accepted from the host without truncation taking place. The default is no truncation. Note the UNIX system has a default 1 megabyte file size limit.
3. *badjobs*—This subfield specifies what to do with undeliverable returning jobs. If an output file is undeliverable for any reason other than file system space limitations (e.g., missing or invalid "usr=" card) and this subfield contains the letter y, the output will be retained in the *job* subdirectory of the home directory; and login *rje* is notified via *mail*(1). If this subfield has any other value, undeliverable output will be discarded. The default is "n".
4. *console*—This subfield specifies the status of the interactive status terminal for this line. If the subfield contains an i, the status console facilities of *rjestat* will be inhibited. In all cases, the normal noninteractive uses of *rjestat* will continue to function. The default is "y".
5. *dial-up*—This subfield contains a telephone number to be used to call a host machine. The telephone number may contain the digits 0 through 9, and the character "-", which denotes a pause. If the telephone number is not present, no dialing is attempted; and a leased line is assumed.

When multiple readers have been specified, jobs that are submitted for transmission to IBM are assigned to the reader with the fewest cards on it. Each reader gets an equal amount of service. This prevents smaller jobs from having to wait for a previously submitted large job to be transmitted. When multiple printers or punches have been specified, returning jobs get assigned to free printers (or punches) allowing smaller output files to bypass large output files.

Deciding how many peripherals to specify depends on the use of that RJE subsystem. If an RJE subsystem is heavily used for off-line printing (i.e., output does not return to the UNIX operating system machine), the administrator would want to specify multiple readers but would not have a need for multiple printers or punches.

## DIRECTORY STRUCTURES

### A. Controlling Directory

The controlling directory used by RJE is */usr/rje*. This directory contains RJE programs for use by separate RJE subsystems (e.g., *rje1*, *rje2*, *rje3*) and the shell queuer's directory. Most RJE programs existing here have been compiled such that each RJE subsystem shares the text of these programs. A snapshot of this directory on our hypothetical machine is as follows:

```
-rwxr-xr-x  3 rje  rje  4068 Mar  4 10:42 cvt
-rw-r--r--  1 rje  rje    42 Apr 10 09:52 lines
-rwxr-xr-x  3 rje  rje 15096 Apr 10 13:01 rjedis
-rwxr-xr-x  3 rje  rje  2328 Mar  4 10:21 rjehalt
-rwxr-xr-x  3 rje  rje 10396 Apr 15 10:07 rjeinit
-r-x-----  3 rje  rje   785 Apr  8 09:00 rjeload
```



```

-rwsr-xr-x 3 rje rje 5040 Mar 27 09:28 rjeqr
-rwxr-xr-x 3 rje rje 4072 Apr 1 15:40 rjerecv
-rwxr-xr-x 3 rje rje 3888 Mar 27 09:35 rjexmit
-rwsr-xr-x 1 root rje 2696 Mar 27 14:42 shqer
-rwxr-xr-x 3 rje rje 5920 Apr 2 15:47 snoop
drwxr-xr-x 2 rje rje 80 Mar 25 13:26 sque

```

The RJE subsystems are generated in their own directory by linking the program names in this directory to the appropriate names in the subsystem directory. The programs are described in the part "RJE PROGRAMS". The file *lines* is the configuration file used by all RJE subsystems. The directory *sque* is used by the shell queuer (*shqer*). This directory contains:

```

-rw-r--r-- 1 rje rje 0 Feb 14 14:04 errors
-rw-r--r-- 1 rje rje 0 Feb 14 14:04 log

```

When *shqer* has work to do, the files *log* and *errors* will be of nonzero length; and temporary files (*tmp\**) will also appear here.

#### B. Subsystem Directory

The RJE subsystem described in this section maintains the connection between *pwba* and IBM *B* and will be referred to as *rje1*. The first line of */usr/rje/lines* describes *rje1*. As noted in this file, *rje1* runs in the directory */usr/rje1*. A snapshot of this directory is as follows:

```

-rw-r--r-- 1 rje rje 4990 Apr 15 08:30 acctlog
-rwxr-xr-x 3 rje rje 4068 Mar 4 10:42 cvt
-rw-r--r-- 1 rje rje 0 Apr 15 04:02 errlog
drwxrwxrwx 2 rje rje 192 Apr 10 09:51 job
-rw-r--r-- 1 rje rje 194 Apr 15 08:11 joblog
-rw-r--r-- 1 rje rje 0 Apr 15 08:11 resp
-rwxr-xr-x 3 rje rje 15096 Apr 10 13:01 rjeldisp
-rwxr-xr-x 3 rje rje 2328 Mar 4 10:21 rjelhalt
-rwxr-xr-x 3 rje rje 10396 Apr 15 10:07 rjelinit

```



|            |       |     |                   |          |
|------------|-------|-----|-------------------|----------|
| -r-x-----  | 3 rje | rje | 785 Apr 8 09:00   | rjelload |
| -rwsr-xr-x | 3 rje | rje | 5040 Mar 27 09:28 | rjelqer  |
| -rwxr-xr-x | 3 rje | rje | 4072 Apr 1 15:40  | rjelrecv |
| -rwxr-xr-x | 3 rje | rje | 3888 Mar 27 09:35 | rjlxmit  |
| drwxr-xr-x | 2 rje | rje | 144 Apr 15 08:30  | rpool    |
| -rwxr-xr-x | 3 rje | rje | 5920 Apr 2 15:47  | snoop0   |
| drwxrwxrwx | 2 rje | rje | 176 Apr 10 13:03  | spool    |
| drwxr-xr-x | 2 rje | rje | 224 Apr 10 13:56  | squeue   |
| -rw-r--r-- | 1 rje | rje | 0 Apr 15 10:30    | stop     |
| -rw-r--r-- | 1 rje | rje | 274 Mar 7 20:25   | testjob  |

The programs **rjel\***, **cvt**, and **snoop0** are linked to the corresponding programs in */usr/rje*. The remaining files and their uses are as follows:

- **acctlog**—Accounting data is stored in this file if it exists. This file is the responsibility of the RJE administrator.
- **errlog**—Used by **rjel** to log errors. It can be useful for debugging **rjel** problems.
- **joblog**—Used by **rjelqer** and **rjestat** to notify **rjlxmit** that a job (or console request) has been submitted. It also contains the process-group number of the **rjel** processes. The program **cvt** can be used to convert this file to a readable form.
- **resp**—Contains console messages received from IBM *R*. These messages can be responses for **rjestat** or IBM responses to submitted jobs (i.e., on reader messages). This file is truncated if it grows to a size greater than 70,000 bytes.
- **stop**—Indicates that **rjelhalt** has been executed. The existence of this file indicates to **rjestat** that **rjel** has been halted by the operator.
- **testjob**—A sample job that can be submitted to test the **rjel** subsystem. Originally, the job control statements may have to be changed to suit your IBM system.

When **rjel** terminates abnormally, the file **dead** should appear in this directory. This file contains a short message indicating why **rjel** is not operating and is used by **rjestat** to report the problem. The remaining directories and their uses are as follows:

- **job**—Used to save undeliverable jobs if the proper parameter has been specified in */usr/rje/lines*. The sample job described above is also delivered to this directory. This directory should be mode 777.
- **rpool**—Contains temporary files used to gather output from the remote machine. These files are named **pr\*** (for print output files) and **pu\*** (for punch output files). Once a complete file has been received, the file is dispatched in the proper way by **rjeldisp**.
- **spool**—Used by **send** to store temporary files to be submitted to the remote machine. This directory must be mode 777.



- *squeue*—Used by *rje1* to store submitted files until they are transmitted. The program *rje1qer* is used by *send* to move the temporary files in the *spool* directory to this directory.

## RJE PROGRAMS

All programs described below, with the exception of *rjestat*, exist in */usr/rje*. These programs are "shared text" and are linked (except *shqer*) to the proper names in each subsystem directory. The names described below are generic; the programs in the *rje2* directory would be *rje2qer*, *rje2init*, etc.

Each available RJE subsystem occupies three process slots. The slots used are *rje?xmit* for the transmitter, *rje?recv* for the receiver, and *rje?disp* for the dispatcher. One additional process slot is used for *shqer* regardless of how many subsystems are available.

Each RJE subsystem tries to be self-sustaining and logs any errors encountered during normal operation in its *errlog* file.

### A. Rjeqer

This program is used by *send* to queue files for transmission. When invoked, it performs the following steps:

1. Moves temporary *pnch(4)* format file in *spool* directory to *squeue* directory.
2. Writes an entry at end of file *joblog* containing:
  - name of file to be transmitted
  - submitter's user ID
  - number of card images in file
  - message level for this job.

The file *joblog* is used to notify *rjexmit* of work to be done.

3. Notifies user that file has been queued.

*Send* determines which host system is desired and invokes the proper *rje?q*er by getting the *prefix* from the *lines* file (e.g., if sending to IBM C from our machine, *rje2qer* would be invoked).

### B. Rjeload

This program is used to start an RJE subsystem. Its prefix determines the subsystem to start (e.g., *rje2load* starts *rje2*). To start the RJE subsystems on UNIX operating system machines, the following commands are executed in */etc/rc* when changing to *init* state 2 (multiuser):

```
rm -f /usr/rje/sque/log
su rje -c " /usr/rje1/rjelload device0 "
su rje -c " /usr/rje2/rje2load device1 "
[If the V.35 interface (see vpmstart(1C) for 3B-20 only)
is used for, say, subsystem rje1,
then use command line:
su rje -c " V_35=1 /usr/rje1/rjelload device0 " ]
```



The file */usr/rje/sque/log* is removed to ensure the correct operation of *shqr*. When invoked, *rjeload* performs the following steps:

1. Uses VPM device from */usr/rje/lines* to link the proper devices [see *vpmset(1C)*].
2. Loads device given as argument with RJE protocol script.
3. Executes *rje?init* to start *rje?* processes (e.g., *rje2load* executes *rje2init*).

#### C. Rjehalt

This program is used to halt an RJE subsystem. To halt *rje2* on UNIX operating system machines, */usr/rje2/rje2halt* is executed. This should be done in the **shutdown** procedure for your machine to ensure graceful termination of RJE. *Rjehalt* will allow only those users with permission to halt an RJE subsystem. *Rjehalt* uses the header on the file *joblog* to get the process-group of the RJE subsystem processes. This group is signaled to terminate. When all processes have terminated, *rjehalt* sends a "signoff" record to the host machine. This signoff record is taken from the file *signoff* (ASCII text) if it exists; otherwise, a "/\*signoff" record is sent. On completion, *rjehalt* creates the file *stop* in the subsystem directory. The presence of the file *stop* in a subsystem directory causes *rjestat* to report to users that RJE to the corresponding host has been stopped by the operator.

#### D. Rjeinit

This program initializes an RJE subsystem. It is used by *rjeload* and can be used to restart a subsystem if the VPM script has previously been started. *Rjeinit* should only be executed by user *rje*. *Rjeinit* fails if there are less than 100 blocks or 10 inodes free in the file system. It issues a warning if there are less than 1.5X blocks (where X is the first field in the parameters for that line) or 100 inodes free in the file system. If *rjeinit* fails, the reason for the failure is reported; and the file *dead* is created containing "Init failed". This will be reported by *rjestat* until a subsequent *rjeinit* succeeds. The *rjeinit* performs the following functions:

1. Dials a remote host if specified.
2. Truncates console response file *resp*.
3. Sends a signon record to the host. The signon record is taken from file *signon* (ASCII text) if it exists; otherwise, *rjeinit* sends a blank record as a signon.
4. Sets up pipes for process communication.
5. Resets process-group for RJE subsystem and restarts error logging.
6. Rebuilds *joblog* file from jobs queued for transmission.
7. Notifies *rjedisp* (via a pipe) of any returned files still remaining in *rpool* directory.
8. Starts appropriate background processes *rje?xmit*, *rje?recv*, and *rje?disp*.
9. Reports started or not started.

If failure occurs in a background process, it is reported by that process (error logging). The failing process will normally attempt to reboot the subsystem by executing *rje?init* with a + as its argument. When *rjeinit* is executed with + as its argument, this indicates an attempted reboot; and *rjeinit* will behave differently.

#### E. Rjexmit

This program writes data to the VPM device. The *rjexmit* is started by *rjeinit* and runs in the background. When running, *rjexmit* performs the following processing:

1. Checks *joblog* file for files to be transmitted. This is done every 5 seconds when not transmitting data. When transmitting data, the *joblog* is checked after transmitting one block from each active reader and



*console.* (*Reader* refers to the logical readers used by RJE. *Console* refers to the RJE logical console which is separate from the logical readers.)

2. Queues files from *joblog* according to first two characters of the file name:
  - *rd\**—These files are queued on the reader with the fewest cards. Normal use of the **send** command creates these files.
  - *sq\**—These files are queued on the last available reader to assure sequential transmission. Using the **-x** option to the **send** command creates these files.
  - *co\**—These files are queued on the console. The **rjstat** command creates these files. All files described above contain EBCDIC data.
3. Sends information to **rjdisp** (via a pipe) for use in user notification of job status.
4. Builds blocks for transmission from active readers and the console. These blocks are built according to the multileaving protocol.
5. Performs following peripheral control:
  - Sends requests to open readers when jobs have been assigned to them. These readers are not active until a grant is received from **rjrecv** (via a pipe).
  - Halts and activates readers when waits or starts (respectively) are received from **rjrecv**.
  - Sends printer or punch grants when an open request is received from **rjrecv**.
6. Notifies **rjdisp** that a file has been transmitted and unlinks the file.

If **rjexmit** encounters fatal errors, it creates the *dead* file with an appropriate message and signals the other background processes to exit. If possible, **rjexmit** will attempt to reboot the RJE subsystem by executing **rjeinit**.

#### F. Rjrecv

This program reads data from the VPM device. The **rjrecv** is started by **rjeinit** and runs in the background. When running, **rjrecv** performs the following processing:

1. Reads blocks of data received from host system.
2. Handles data received according to its type. The two types of data are:
  - **Control information**—**Rjrecv** performs the following peripheral device control:
    - a. Notifies **rjexmit** of grants to its requests to open readers.
    - b. Passes wait and start reader information to **rjexmit**.
    - c. Passes open requests (for printers and punches) from the host to **rjexmit**.
  - **User Information**—The three major types of user information received are:
    - a. Console responses and job status messages. This data is appended to the *resp* file for use by **rjstat** and **rjdisp**.



- b. The printer output from user jobs. This data is collected in temporary files (*pr\**) in the *rpool* directory. When a complete print job has been received, **rjerecv** notifies **rjedis** (via a pipe) that the file is to be dispatched.
  - c. The punch output from user jobs. This data is handled the same as printer output except that the *rpool* files are named *pu\**.
3. If console response file *resp* exceeds 70,000 characters, **rjerecv** truncates file.
4. **Rjerecv** stops accepting output from the remote machine if the number of free blocks in the file system falls below *space* blocks.
5. **Rjerecv** truncates files to *size* blocks if a received file exceeds this value.

If **rjerecv** encounters fatal errors, it creates the *dead* file with an appropriate error message, signals the other background processes to exit, and reboots the RJE subsystem.

#### G. Rjedis

This program dispatches user information. **Rjedis** is started by **rjeinit** and runs in the background. When running, **rjedis** performs the following processing:

1. Dispatches output. The two types of output are printer and punch output. After receiving notification of output ready from **rjerecv**, **rjedis** searches for a "usr=" line in the received file. The format of a "usr=" line is as follows:

usr=(user,place,level)

**Rjedis** dispatches output according to the place field.

2. Dispatches messages. The three types of messages are as follows:
  - Job transmitted—This message is sent to the submitting user when **rjedis** reads this event notice from the **rjexmit** pipe.
  - Job acknowledgment—**rjedis** dispatches IBM acknowledgment messages to submitting users.
  - Output processing—**rjedis** dispatches job output messages according to the options specified on the "usr=" card. A normal output message indicates the returned file name is ready. Messages can be masked by using the *level* on the "usr=" card.
3. Whenever output is to be handled by **shqer**, **rjedis** checks that **shqer** is running. This is done by looking for the **shqer log** file. If this file does not exist, **rjedis** starts **shqer**.

#### H. Shqer

This program executes user programs when they appear in the *place* field of the "usr=" line in a returned output file (print or punch). **Shqer** is started by **rjedis** when the first output file using this feature is returned. Subsequent files using this feature are logged for execution by **rjedis**. When started, **shqer** performs the following processing:

1. Builds the *log* file from file names in */usr/rje/sque* directory. Each log entry is the name of a file (*tmp?*) that contains the following information:
  - name of file to be executed



- name of input file (file returned from IBM)
  - name of IBM job
  - programmer's name
  - IBM job number
  - user's name from "usr=" line
  - user's login directory
  - minimum file system space.
2. **Shqer** uses two parameters. The first is the delay time between *log* file reads. The second is a *nice*(2) factor which is applied to any programs spawned by **shqer**. These values are defined in */usr/include/rje.h* (*QDELAY* and *QNICE*).
3. When each log entry is read, the appropriate program is spawned with the following characteristics:
- The returned RJE file is standard input to the program.
  - The standard and diagnostic outputs are */dev/null*.
  - The *LOGNAME*, *HOME*, and *TZ* variables are set to appropriate values.
  - The arguments to the spawned program, in order, are:
    - a. a numerical value indicating that file system free space is equal or above (0) or below (1) *space* blocks.
    - b. IBM job name.
    - c. programmer's name.
    - d. IBM job number.
    - e. user's login name.
4. After executing each program, the *tmp?* file and returned RJE file are removed.

#### UTILITY PROGRAMS

##### A. Snoop

**Snoop** is the generic name of a program that can be used to trace the state of a VPM device and its associated communications line. **Snoop** depends on the *trace*(7) driver for its information. It reads trace entries from */dev/trace* and converts them into a readable form that is printed on the standard output.

The usable name of **snoop** for a particular RJE subsystem is **snoopN**, where *N* is the minor device number of the VPM device. In our hypothetical system, *vpm0* is used by the *rje1* subsystem; and *vpml* is used by the *rje2* subsystem. Therefore, */usr/rje1/snoop0* and */usr/rje2/snoop1* are linked to */usr/rje/snoop*.



Each **snoop** prints trace entries for its associated VPM device. Trace entries are printed in the following form:

| <i>sequence</i> | <i>type</i> | <i>information</i> |
|-----------------|-------------|--------------------|
|-----------------|-------------|--------------------|

where:

- *sequence* specifies the order of trace occurrences. It is a value between 0 and 99.
- *type* specifies the action being traced (e.g., transfers, driver activity).
- *information* describes data being transferred and driver activity.

Refer to Table 10.A for the meaning of the trace *types* and associated *information*.

#### B. Rjestat

This program is supplied as a user command. The program's two functions are to describe the status of the RJE subsystems and to provide a remote IBM status console. The remainder of this part describes these two functions.

##### RJE Status

When invoked, **rjestat** reports the status of the RJE subsystems. If remote system ("host") names are specified, only those statuses are reported. The **rjestat** uses the following rules to report the status of a subsystem:

- **Rjestat** prints the contents of the file *status* if it exists in the subsystem directory. This file can contain any message the administrator wishes to have printed when users use **rjestat**.
- If the file *dead* exists in the subsystem's directory, the subsystem is not operating and the reason is contained in the file. The **rjestat** reports that RJE to "host" is down and prints the contents of the *dead* file as the reason.
- If the file *stop* exists in the subsystems directory, the **rjehalt** program has been used to inhibit that RJE subsystem. **Rjestat** reports that RJE to "host" has been stopped by the operator.
- If neither the *dead* nor the *stop* file exists, **rjestat** reports that RJE to "host" is operating normally.

**Rjestat** is supplied as the user's vehicle for checking the status of RJE. It is not meant to be an administrative tool; however, the reason for failure can be used to track the problem.

##### Status Console

To use **rjestat** as a status console, the **-s host** argument is used. **Rjestat** prints the status of the subsystem, then prompts with *host*: if the subsystem is up. Each console request is submitted to the RJE processes for transmission, and output is handled as specified. **Rjestat** checks the status prior to submitting each request and will tell the user to try later if the subsystem goes down. **Rjestat** allows the rje or superuser logins to submit other than display requests. For a complete description of how to use the status console features, see **rjestat(1C)**.

#### C. Cvt

This program converts any subsystem's *joblog* file to readable form. The first line printed is the process group number of the subsystem processes. The remaining output consists of entries in the following form:

| <i>file</i> | <i>user-id</i> | <i>records</i> | <i>level</i> |
|-------------|----------------|----------------|--------------|
|-------------|----------------|----------------|--------------|

where "file" is the name of the submitted file, "user-id" is the submitters user number, "records" is the number of card images, and "level" is the message level. The "records" and "level" fields are not used if the file name is *co\** (console request submitted by **rjestat**).



## RJE ACCOUNTING

Each RJE subsystem will store accounting information in the *acctlog* file if it exists. It is the responsibility of the RJE administrator to create and maintain this file in the subsystem's directory. Entries in this file describe RJE line use and are of the following form:

| day | time | file | user | records |
|-----|------|------|------|---------|
|-----|------|------|------|---------|

Each field is delimited by a tab character. The meanings of each field is as follows:

1. day—The day of occurrence in the form *mm/dd*.
2. time—The time of occurrence in the form *hh:mm:ss*.
3. file—The name of the UNIX system file. The first two characters identify its type as follows:
  - *rd/sq*—The file was transmitted to the remote system.
  - *pr*—The print output file was received from the remote system.
  - *pu*—The punch output file was received from the remote system.
4. user—The user ID of the user responsible for the transfer.
5. records—The number of records (card images) transferred for this file.

Since *acctlog* data is not used by RJE, it should not be allowed to grow too large. This can be accomplished by moving or processing the file during a system reboot (i.e., in */etc/rc* before the RJE subsystems are started).

The following list describes some of the reports that could be generated from the *acctlog* data. Implementation of a program to produce accounting reports is the responsibility of the administrator.

- Periodic Reports—By using the "day" and "time" fields in the data, periodic usage reports can be produced.
- By User Reports—By using the "user" field in the data, usage-by-user reports can be produced.
- By Subsystem Reports—By using the */usr/rje/lines* file information and each *acctlog* file, a usage-by-subsystem (or remote system) report can be produced. Other reports can be produced using the type of file, size of jobs, etc.

## TROUBLESHOOTING

This section deals with RJE problems and some methods for resolving them. The topics discussed in this section are as follows:

- Automatic Error Recovery
- Manual Error Recovery
- RJE Problems
- VPM Problems
- Trace Interpretation.



### A. Automatic Error Recovery

RJE attempts to be self-sustaining with respect to its availability. In general, if problems occur on the communications line or the remote machine (e.g., a crash), RJE will continually try to restart itself (this action will be referred to as "reboot"). For example, if an RJE subsystem is started using `rjeload` but the IBM system is not available, a fatal error will occur. The process that detects this error (usually `rjexmit` or `rjerecv`) will reboot the subsystem by executing `rjeinit` with a `+` as its argument. When `rjeinit` detects a `+` argument, it waits 1 minute before attempting to bring up the subsystem.

The `rjehalt` program can be used to prevent an RJE subsystem from rebooting itself when the remote system is not available for a known period of time. When the remote system is made available, the subsystem may be started in the normal way.

### B. Manual Error Recovery

In order to manually recover from errors, one must know how to start and stop an RJE subsystem. There are two ways to start an RJE subsystem:

- `rje?load`—This program loads and starts the VPM script and executes `rje?init`.
- `rje?init`—This program starts the `rje?` subsystem. In order to use this program, the VPM script must be loaded and started.

To stop the `rje?` subsystem, the `rje?halt` program should be executed. This stops the subsystem gracefully and will prevent a reboot.

The `rjeload` program must be used to start RJE for the first time (after a UNIX system reboot). Subsequently, as long as the script is running, execution sequences of `rjehalt` and `rjeinit` will stop and start RJE.

Manually starting and stopping RJE can be useful in tracking down problems. For example, if user jobs are not being submitted to the host machine, the following sequence can ease identification of the problem:

1. Halt ailing subsystem.
2. Start a `snoop` process in the background with its output redirected to a file.
3. Restart subsystem.
4. Scan `snoop` output to determine where the problem is.

The `snoop` program is the most useful software tool for identifying RJE problems. Its uses are described in the subpart "Trace Interpretation".

### C. RJE Problems

This section describes problems that can occur in an RJE subsystem. These problems generally occur when the subsystem has not been set up properly. The following is a list of things to check to ensure that an RJE subsystem has been set up properly.

1. IBM description—The description of the remote UNIX machine must be consistent with the description in the subpart "IBM Generation".
2. UNIX system description—The file `/usr/rje/lines` must be set up properly. The subpart "UNIX System Generation" describes this file in detail.



3. VPM setup—The VPM software must be installed and the proper VPM and physical devices made. Each VPM device must correspond to the proper physical device; see `vpm(7)`.
4. Free space—As a general rule, all file systems must have a reasonable amount of free space. File systems containing RJE subsystems must have sufficient free space to ensure proper RJE operation.
5. Directories—Each subsystem's directory and the controlling directory should be checked for the following:
  - All needed files exist.
  - The proper prefix is on each applicable RJE program.
  - The link count is correct for files that are linked.
  - All file and directory modes are correct.
6. Initialization—Peripherals information must be consistent on both systems. The line must be started on the IBM system, proper hardware connections made, etc.

Problems with a subsystem are indicated by error messages. The `rjeinit` checks for obstacles in bringing up RJE. If an obstacle is found, an error message indicating the obstacle is printed on the error output. If a problem is encountered during normal operation, the message is logged in the `errlog` file. This file, error messages, output from `snoop`, and the checklist above should be used to determine and fix any subsystem problems. Generally, if a subsystem is set up properly but will not operate, the problem is the way the VPM or KMC has been set up, the remote system, or the hardware.

#### D. VPM Problems

After installing the hardware and making the appropriate devices, all VPM software and devices must be made [see `vpm(7)`]. The program `rjeload` links the devices to be used by the corresponding RJE subsystem.

The following is a list of items to check when problems occur:

1. Proper hardware—The appropriate hardware must be installed. Be sure the device is properly described to the system and passes diagnostics.
2. Proper devices—The major and minor device numbers for the physical device and VPM devices must be correct. It should also be verified that `rjeload` program is called with the correct device names.
3. Script runs—Verify the VPM script is able to run. This is done by tracing the proper device with the proper `snoop` program. `Snoop` will print "started" entries for both the physical device and VPM script. If no output appears from `snoop` when `rjeload` is executed, either the hardware is not working properly or the hardware or VPM has not been set up properly. Output of any other type from `snoop` should indicate where the problem is occurring.

#### E. Trace Interpretation

This part describes how to interpret trace output from the `snoop` program and gives several examples.

Lines with type TR are traces from the VPM script. All others are driver traces and indicate the following:

- CL—Activity occurring when the device has been closed.
- OP—Activity occurring when the device has been opened.



- RD—Read from device occurred.
- WR—Write to device occurred.
- ST—Start or stop activity.
- SC—Script termination type, termination value is given.

Table 10.A enumerates all possible trace lines for each type and describes the event. The remainder of this part consists of example trace output and its interpretation. Comments describing events will appear after the "\*" in trace output. If more than one VPM were running, sequence numbers might not appear in order. For clarity, example sequences will be in order.

#### Normal RJE Startup

The following is an example of trace output when RJE has been started up. In this case the remote machine responds to the enquiry byte (ENQ). The RJE subsystem signs on to the machine then follows the handshaking protocol (exchanging ACKs).

| Tracing device0 |    |          |                           |
|-----------------|----|----------|---------------------------|
| 0               | ST | Startack | * Physical device started |
| 1               | TR | Started  | * Script started          |
| 2               | ST | Start    | * VPM Driver start        |
| 3               | OP | Opened   | * VPM Device open         |
| 4               | WR | 84 bytes | * Signon record written   |
| 5               | TR | S-ENQ    | * Enquiry byte sent       |
| 6               | TR | R-ACK    | * Received acknowledgment |
| 7               | TR | S-BLK    | * Sent signon block       |
| 8               | TR | R-ACK    | * Block acknowledged      |
| 9               | TR | S-ACK    | * Handshaking             |
| 10              | TR | R-ACK    | * .                       |
| 11              | TR | S-ACK    | * .                       |
| 12              | TR | R-ACK    | * .                       |
| 13              | TR | S-ACK    | * .                       |
| 14              | TR | R-ACK    | * .                       |
| 15              | TR | S-ACK    | * .                       |
| 16              | TR | R-ACK    | * .                       |
| 17              | TR | S-ACK    | * Handshaking             |

If any jobs had been submitted via the **send** command or jobs were waiting to be returned, the traces would reflect the transfers rather than handshaking.

#### RJE Startup—IBM Not Responding

This example shows trace output when RJE has been started but does not receive a response from the remote machine. In general, the RJE script will time-out if a response is not received from the remote machine within 3 seconds of the last transmission. When a time-out is detected while starting up, the ENQ is retransmitted. This is repeated six times before the script gives up. Other time-out responses will be discussed later.



## Tracing device0

|    |    |            |   |                           |
|----|----|------------|---|---------------------------|
| 86 | ST | Startack   | * | Physical device started   |
| 87 | TR | Started    | * | Script started            |
| 88 | ST | Start      | * | VPM driver start          |
| 89 | OP | Opened     | * | VPM device open           |
| 90 | WR | 84 bytes   | * | Signon record written     |
| 91 | TR | S-ENQ      | * | Enquiry byte sent         |
| 92 | TR | TIME-OUT   | * | No response to enquiry    |
| 93 | TR | S-ENQ      | * | Enquiry byte sent         |
| 94 | TR | TIME-OUT   | * | No response               |
| 95 | TR | S-ENQ      | * | Enquiry byte sent         |
| 96 | TR | TIME-OUT   | * | No response               |
| 97 | TR | S-ENQ      | * | Enquiry byte sent         |
| 98 | TR | TIME-OUT   | * | No response               |
| 99 | TR | S-ENQ      | * | Enquiry byte sent         |
| 0  | TR | TIME-OUT   | * | No response               |
| 1  | TR | S-ENQ      | * | Enquiry byte sent         |
| 2  | TR | TIME-OUT   | * | No response               |
| 3  | RD | 1 bytes    | * | 1 byte read (error)       |
| 4  | ST | Stopchk    | * | Safety check              |
| 5  | ST | Stopack(0) | * | Script termination normal |
| 6  | CL | Clean      | * | Cleanup done              |
| 7  | ST | Stopped    | * | VPM script stopped        |
| 8  | CL | Closed     | * | VPM device closed         |

The above sequence will be repeated approximately every minute until a positive response is received from the host. During that minute, the RJE subsystem is dormant; and the `rjstat` command will report that IBM is not responding. When this occurs, either the IBM machine is not available, down, line not started, etc.; or there is a communications problem somewhere from where the physical device transmits data to where it receives data. The RJE administrator should first verify that the IBM machine is up, and the communications line has been started. If so, a hardware trace of the communications line should be done to aid in detecting the problem.

**Transmitting and Receiving**

This example shows trace output from the start of job transmission through its return. For simplicity, only one job is being transmitted and returned.



|                 |    |           |                                  |
|-----------------|----|-----------|----------------------------------|
| Tracing device0 |    |           |                                  |
| 94              | TR | R-ACK     | * Handshaking                    |
| 95              | TR | S-ACK     | * .                              |
| 96              | TR | R-ACK     | * .                              |
| 97              | TR | S-ACK     | * Handshaking                    |
| 98              | WR | 4 bytes   | * Open reader request written    |
| 99              | TR | R-ACK     | * Handshaking                    |
| 0               | TR | S-BLK     | * Sent open request block        |
| 1               | TR | R-OKBLK   | * Received block (grant)         |
| 2               | TR | S-ACK     | * Block acknowledged             |
| 3               | RD | 7 bytes   | * Read seven bytes (grant)       |
| 4               | TR | R-ACK     | * Handshaking                    |
| 5               | TR | S-ACK     | * Handshaking                    |
| 6               | WR | 481 bytes | * First block written            |
| 7               | WR | 470 bytes | * Second block written           |
| 8               | TR | R-ACK     | * Handshaking                    |
| 9               | TR | S-BLK     | * First block sent               |
| 10              | TR | R-ACK     | * Block acknowledged             |
| 11              | WR | 470 bytes | * Third block written            |
| 12              | TR | S-BLK     | * Second block sent              |
| 13              | TR | R-OKBLK   | * Received block (on reader msg) |
| 14              | WR | 470 bytes | * Fourth block written           |
| 15              | RD | 66 bytes  | * Read 66 bytes (on reader msg)  |
| 16              | TR | S-BLK     | * Third block sent               |
| 17              | TR | R-ACK     | * Block acknowledged             |
| 18              | WR | 147 bytes | * Fifth block written            |
| 19              | TR | S-BLK     | * Fourth block sent              |
| 20              | TR | R-ACK     | * Block acknowledged             |
|                 |    | .         | *                                |
|                 |    | .         | *                                |
|                 |    | .         | * More of the same               |
|                 |    | .         | *                                |
| 93              | TR | R-ACK     | * Handshaking                    |
| 94              | TR | S-ACK     | * Handshaking                    |
| 95              | TR | R-OKBLK   | * Received block (request)       |
| 96              | TR | S-ACK     | * Block acknowledged             |
| 97              | RD | 7 bytes   | * Read open printer request      |
| 98              | TR | R-ACK     | * Handshaking                    |
| 99              | TR | S-ACK     | * .                              |
| 0               | TR | R-ACK     | * .                              |
| 1               | TR | S-ACK     | * .                              |
| 2               | TR | R-ACK     | * .                              |



|    |    |           |                         |
|----|----|-----------|-------------------------|
| 3  | TR | S-ACK     | * Handshaking           |
| 4  | WR | 4 bytes   | * Printer grant written |
| 5  | TR | R-ACK     | * Handshaking           |
| 6  | TR | S-BLK     | * Block sent (grant)    |
| 7  | TR | R-OKBLK   | * First block received  |
| 8  | TR | S-ACK     | * Block acknowledged    |
| 9  | RD | 64 bytes  | * Read first block      |
| 10 | TR | R-OKBLK   | * Second block received |
| 11 | TR | S-ACK     | * Block acknowledged    |
| 12 | RD | 505 bytes | * Read second block     |
| 13 | TR | R-OKBLK   | * Third block received  |
| 14 | TR | S-ACK     | * Block acknowledged    |
| 15 | TR | R-OKBLK   | * Fourth block received |
| 16 | TR | S-ACK     | * Block acknowledged    |
| 17 | TR | R-ACK     | * Handshaking           |
| 18 | TR | S-ACK     | * .                     |
| 19 | TR | R-ACK     | * .                     |
| 20 | TR | S-ACK     | * Handshaking           |
| 21 | RD | 470 bytes | * Read third block      |
| 22 | RD | 494 bytes | * Read fourth block     |
| 23 | TR | R-ACK     | * Handshaking           |
| 24 | TR | S-ACK     | * Handshaking           |
|    | .  |           | * .                     |
|    | .  |           | * .                     |
|    | .  |           | * etc.                  |
|    | .  |           | * .                     |

Requests and grants are part of the multileaving protocol. When jobs are being transmitted and received simultaneously, as in a busier RJE subsystem, much less handshaking is involved. Rather than acknowledging blocks with ACKs, the protocol allows a block to be returned (this implies acknowledgment of the received block). The following example shows trace output at a busy time:

| tracing device0 |    |           |                  |
|-----------------|----|-----------|------------------|
| 45              | TR | R-OKBLK   | * Received block |
| 46              | TR | S-BLK     | * Sent block     |
| 47              | WR | 493 bytes | *                |
| 48              | RD | 496 bytes | *                |
| 49              | TR | R-OKBLK   | * Received block |
| 50              | RD | 65 bytes  | *                |
| 51              | WR | 4 bytes   | *                |
| 52              | TR | S-BLK     | * Sent block     |
| 53              | TR | R-OKBLK   | * Received block |
| 54              | TR | S-BLK     | * Sent block     |
| 55              | WR | 493 bytes | *                |
| 56              | RD | 7 bytes   | *                |
| 57              | TR | R-OKBLK   | * Received block |
| 58              | WR | 493 bytes | *                |
| 59              | RD | 496 bytes | *                |
| 60              | TR | S-BLK     | * Sent block     |
| 61              | TR | R-OKBLK   | * Received block |



Notice that since there is work to be done on both sides acknowledgments are implied.

#### Time-out Error Recovery

This example shows activity resulting from time-outs occurring during normal operation. These time-outs were caused because the remote JES3 system has performance problems, and occasionally, does not respond in the required 3 seconds.

#### Time-Out Error Recovery

##### Tracing device1

|    |    |          |                    |
|----|----|----------|--------------------|
| 27 | TR | S-ACK    | * Handshaking      |
| 28 | TR | R-ACK    | * .                |
| 29 | TR | S-ACK    | * .                |
| 30 | TR | TIME-OUT | * No response      |
| 31 | TR | S-NAK    | * Not acknowledged |
| 32 | TR | TIME-OUT | * No response      |
| 33 | TR | S-NAK    | * Not acknowledged |
| 34 | TR | R-ACK    | * Response         |
| 35 | TR | S-ACK    | * Handshaking      |
| 36 | TR | R-ACK    | * .                |
|    | .  |          | * .                |
|    | .  |          | * .                |
|    | .  |          | * .                |
| 54 | TR | R-ACK    | * .                |
| 55 | TR | S-ACK    | * Handshaking      |
| 56 | TR | TIME-OUT | * No response      |
| 57 | TR | S-NAK    | * Not acknowledged |
| 58 | TR | R-ACK    | * Response         |
| 59 | TR | S-ACK    | * Handshaking      |
|    | .  |          |                    |
|    | .  |          |                    |

The response to these time-outs are NAKs (not acknowledged). RJE will respond this way up to six times before giving up and attempting a reboot. At this time **rjestat** would report that there are "Line Errors". NAK is a request to retransmit the previous response.

#### Communication Line Errors

This example shows trace output from an RJE subsystem that uses a dial-up connection. The phone line is noisy and is prone to dropping.



## Tracing vpm1

|    |    |            |   |                           |
|----|----|------------|---|---------------------------|
| 63 | TR | S-ACK      | * | Handshaking               |
| 64 | TR | R-ACK      | * | •                         |
| 65 | TR | S-ACK      | * | •                         |
| 66 | TR | R-JUNK     | * | Noise on the line         |
| 67 | TR | S-NAK      | * | Not acknowledged          |
| 68 | TR | R-ACK      | * | Recovery                  |
| 69 | TR | S-ACK      | * |                           |
| 70 | TR | R-ACK      | * |                           |
| 71 | TR | S-ACK      | * |                           |
| 72 | TR | TIME-OUT   | * | Line has dropped          |
| 73 | TR | S-NAK      | * | Attempting to recover     |
| 74 | TR | TIME-OUT   | * | •                         |
| 75 | TR | S-NAK      | * | •                         |
| 76 | TR | TIME-OUT   | * | •                         |
| 77 | TR | S-NAK      | * | •                         |
| 78 | TR | TIME-OUT   | * | •                         |
| 79 | TR | S-NAK      | * | •                         |
| 80 | TR | TIME-OUT   | * | •                         |
| 81 | TR | S-NAK      | * | •                         |
| 82 | TR | TIME-OUT   | * | •                         |
| 83 | RD | 1 bytes    | * | 1 byte read (error)       |
| 84 | ST | Stopack(0) | * | Script termination normal |
| 85 | CL | Clean      | * | Cleanup                   |
| 86 | ST | Stopped    | * | VPM script stopped        |
| 87 | CL | Closed     | * | VPM device closed         |

The error read in the above sequence causes RJE to reboot and rjestat to report line errors. If this type of thing were to occur frequently, a different method of communication should be used.

#### Error Responses

As seen in the parts above, the response to most errors is to send a NAK. The only exception is when starting up. Whenever a NAK is received on either side, it indicates that the previous transmission was not properly received. This should be followed by retransmission of the previous data. Generally, NAKs should not occur frequently and should be followed by recovery. If errors occur frequently or NAKs do not cause recovery, the line should be checked for problems.

On some IBM systems (e.g., JES2), an I/O error is printed at the system console whenever a NAK is received. These I/O errors can also be helpful in detecting the problem; however, they will not be discussed here as they vary with the system. It is assumed that someone in IBM support can assist if needed.



TABLE 10.A

| TYPE | INFORMATION            | MEANING                                                                                                                                                               |
|------|------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CL   | Closed                 | The VPM device has been closed.                                                                                                                                       |
| CL   | Clean                  | The VPM driver is cleaning up for this device.                                                                                                                        |
| OP   | Opened                 | The VPM has been successfully opened.                                                                                                                                 |
| OP   | Failed (start)         | The open failed because the script within the physical device could not be started.                                                                                   |
| RD   | <i>num</i> bytes       | <i>Num</i> bytes were read from the VPM device by <i>rjerecv</i> .                                                                                                    |
| SC   | Exit ( <i>num</i> )    | The VPM script has terminated abnormally. The VPM termination code is <i>num</i> . Termination codes are defined in <i>vpm</i> (7).                                   |
| ST   | Startack               | The physical device acknowledges that it has been started.                                                                                                            |
| ST   | Stopchk                | Safety wakeup check on stop procedure.                                                                                                                                |
| ST   | Stopack ( <i>num</i> ) | The physical device acknowledges that it is being halted by the vpm device. The VPM termination code is <i>num</i> . Termination codes are defined in <i>vpm</i> (7). |
| ST   | Stopped                | The VPM script has been stopped.                                                                                                                                      |
| TR   | Started                | The script has started tracing.                                                                                                                                       |
| TR   | R-ACK                  | A 2-byte acknowledgment (ACK) string has been received from the remote system. This indicates that the previous transmission was properly received.                   |
| TR   | S-ACK                  | A 2-byte acknowledgment (ACK) string has been transmitted to the remote system.                                                                                       |
| TR   | R-NAK                  | A "not-acknowledged" (NAK) character has been received from the remote system. This indicates that the previous transmission was not properly received.               |
| TR   | S-NAK                  | A "not-acknowledged" (NAK) character has been transmitted to the remote system.                                                                                       |
| TR   | R-ENQ                  | An enquiry (ENQ) character has been received from the remote system.                                                                                                  |
| TR   | S-ENQ                  | An enquiry (ENQ) character has been transmitted to the remote system.                                                                                                 |
| TR   | R-WAIT                 | The remote machine has requested that no data be transmitted to it.                                                                                                   |
| TR   | R-OKBLK                | A valid data block was received from the remote machine.                                                                                                              |
| TR   | R-ERRBLK               | An invalid Cyclic Redundancy Check (CRC) was received with a data block.                                                                                              |
| TR   | R-SEQERR               | The block sequence count on a received data block was invalid.                                                                                                        |
| TR   | R-JUNK                 | An invalid data block was received from the remote system.                                                                                                            |
| TR   | TIME-OUT               | The remote machine did not respond within 3 seconds.                                                                                                                  |
| TR   | S-BLK                  | A data block has been transmitted to the remote system.                                                                                                               |
| WR   | <i>num</i> bytes       | <i>Num</i> bytes were written to the VPM device by <i>rjexmit</i> .                                                                                                   |



# UNIX System

## Activity Package



This document is part of the ADMINISTRATOR'S GUIDE. Therefore the pagenumbers don't begin with 1.

**Trademarks:**

MUNIX, CADMUS  
DEC, PDP  
UNIX

for PCS  
for DEC  
for Bell Laboratories

Copyright 1984 by  
PCS GmbH, Pfälzer-Wald-Strasse 36, D-8000 München 90, tel. (089) 67804-0

The information contained herein is the property of PCS and shall neither be reproduced in whole or in part without PCS's prior written approval nor be implied to grant any license to make, use or sell equipment manufactured herewith.

PCS reserves the right to make changes without notice in the specifications and materials contained herein and shall not be responsible for any damages (including consequential) caused by reliance on the materials presented.



## 11. UNIX SYSTEM ACTIVITY PACKAGE

### General

This section describes the design and implementation of the UNIX System Activity Package. The UNIX operating system contains a number of counters that are incremented as various system actions occur. The system activity package reports UNIX system-wide measurements including Central Processing Unit(CPU) utilization, disk and tape Input/Output(I/O) activities, terminal device activity, buffer usage, system calls, system switching and swapping, file-access activity, queue activity, and message and semaphore activities. The package provides four commands that generate various types of reports. Procedures that automatically generate daily reports are also included. The five functions of the activity package are:

- **sar(1)** command—allows a user to generate system activity reports in real-time and to save system activities in a file for later usage.
- **sag(1G)** command—displays system activity in a graphical form.
- **sadp(1)** command—samples disk activity once every second during a specified time interval and reports disk usage and seek distance in either tabular or histogram form.
- **timex(1)**—a modified **time(1)** command that times a command and also reports concurrent system activity.
- **system activity daily reports**—procedures are provided for sampling and saving system activities in a data file periodically and for generating the daily report from the data file.

The system activity information reported by this package is derived from a set of system counters located in the operation system kernel. These system counters are described in the part "System Activity Counters". The part "System Activity Commands" describes the commands provided by this package. The procedure for generating daily reports is given in "Daily Report Generation". A description for each of the files used by the system activity package can be found in Attachment 11.1.

### System Activity Counters

The UNIX operating system manages a number of counters that record various activities and provide the basis for the system activity reporting system. The data structure for most of these counters is defined in the *sysinfo* structure (see Attachment 11.2) in */usr/include/sys/sysinfo.h*. The system table overflow counters are kept in the *\_syserr* structure. The device activity counters are extracted from the device status tables. In this version, the I/O activity of the following devices is recorded: RP06, RM05, RS04, RF11, RK05, RP03, RL02, TM03, and TM11.

In the following paragraphs, the system activity counters that are sampled by the system activity package are described.

**Cpu time counters:** There are four time counters that may be incremented at each clock interrupt 60 times per second. Exactly one of the *cpu[]* counters is incremented on each interrupt, according to the mode the CPU is in at the interrupt; idle, user, kernel, and wait for I/O completion.

**Lread and lwrite:** The *lread* and *lwrite* counters are used to count logical read and write requests issued by the system to block devices.

**Bread and bwrite:** The *bread* and *bwrite* counters are used to count the number of times data is transferred between the system buffers and the block devices. These actual I/Os are triggered by logical I/Os that



cannot be satisfied by the current contents of the buffers. The ratio of block I/O to logical I/O is a common measure of the effectiveness of the system buffering.

**Phread and phwrite:** The *phread* and *phwrite* counters count read and write requests issued by the system to raw devices.

**Swapin and swapout:** The *swapin* and *swapout* counters are incremented for each system request initiating a transfer from or to the swap device. More than one request is usually involved in bringing a process into memory, or out, because text and data are handled separately. Frequently used programs are kept on the swap device and are swapped in rather than loaded from the file system. The *swapin* counter reflects these initial loading operations as well as resumptions of activity, while the *swapout* counter reveals the level of actual "swapping." The amount of data transferred between the swap device and memory are measured in blocks and counted by *bswapin* and *bswapout*.

**Pswitch and syscall:** These counters are related to the management of multiprogramming. *Syscall* is incremented every time a system call is invoked. The numbers of invocations of *read(2)*, *write(2)*, *fork(2)*, and *exec(2)* system calls are kept in counters *sysread*, *syswrite*, *sysfork*, and *sysexec*, respectively. *Pswitch* counts the times the switcher was invoked, which occurs when:

- a. a system call resulted in a read block
- b. an interrupt occurred resulting in awakening a higher priority process
- c. 1-second clock interrupt.

**Iget, namei, and dirblk:** These counters apply to file-access operations. *Iget* and *namei*, in particular, are the names of UNIX operating system routines. The counters record the number of times that the respective routines are called. *Namei* is the routine that performs file system path searches. It searches the various directory files to get the associated i-number of a file corresponding to a special path. *Iget* is a routine called to locate the inode entry of a file (i-number). It first searches the in-core inode table. If the inode entry is not in the table, routine *iget* will get the inode from the file system where the file resides and make an entry in the in-core inode table for the file. *Iget* returns a pointer to this entry. *Namei* calls *iget*, but other file access routines also call *iget*. Therefore, counter *iget* is always greater than counter *namei*.

Counter *dirblk* records the number of directory block reads issued by the system. It is noted that the directory blocks read divided by the number of *namei* calls estimates the average path length of files.

**Runque, runocc, swpque, and swpocc:** These counters are used to record queue activities. They are implemented in the *clock.c* routine. At every 1 second interval, the clock routine examines the process table to see whether any processes are in core and in ready state. If so, the counter *runocc* is incremented and the number of such processes are added to counter *runque*. While examining the process table, the clock routine also checks whether any processes in the swap device are in ready state. The counter *swpocc* is incremented if the swap queue is occupied, and the number of processes in swap queue is added to counter *swpque*.

**Readch and writech:** The *readch* and *writech* counters record the total number of bytes (characters) transferred by the *read* and *write* system calls, respectively.

**Monitoring terminal device activities:** There are six counters monitoring terminal device activities. *Rcvint*, *xmtint*, and *mdmint* are counters measuring hardware interrupt occurrences for receiver, transmitter, and modem individually. *Rawch*, *canch*, and *outch* count number of characters in the raw queue, canonical queue, and output queue. Characters generated by devices operating in the *cooked* mode, such as terminals, are counted in both *rawch* and (as edited) in *canch*, but characters from raw devices, such as communication processors, are counted only in *rawch*.

**Msg and sema counters:** These counters record message sending and receiving activities and semaphore operations, respectively.



**Monitoring I/O activities:** As to the I/O activity for a disk or tape device, four counters are kept for each disk or tape drive in the device status table. Counter *io\_ops* is incremented when an I/O operation has occurred on the device. It includes block I/O, swap I/O, and physical I/O. *Io\_bcmt* counts the amount of data transferred between the device and memory in 512 byte units. *Io\_act* and *io\_resp* measure the active time and response time of a device in time ticks summed over all I/O requests that have completed for each device. The device active time includes the device seeking, rotating and data transferring times, while the response time of an I/O operation is from the time the I/O request is queued to the device to the time when the I/O completes.

**Inodeovf, fileovf, textovf, and procovf:** These counters are extracted from *\_syserr* structure. When an overflow occurs in any of the inode, file, text and process tables, the corresponding overflow counter is incremented.

### System Activity Commands

The system activity package provides three commands for generating various system activity reports and one command for profiling disk activities. These tools facilitate observation of system activity during

- a controlled stand-alone test of a large system
- an uncontrolled run of a program to observe the operating environment
- normal production operation.

Commands *sar* and *sag* permit the user to specify a sampling interval and number of intervals for examining system activity and then to display the observed level of activity in tabular or graphical form. The *timex* command reports the amount of system activity that occurred during the precise period of execution of a timed command. The *sadp* command allows the user to establish a sampling period during which access location and seek distance on specified disks are recorded and later displayed as a tabular summary or as a histogram.

### The "sar" command

The *sar* command can be used in the following ways:

- When the frequency arguments *t* and *n* are specified, it invokes the data collection program *sadc* to sample the system activity counters in the operating system every *t* seconds for *n* intervals and generates system activity reports in real-time. Generally, it is desirable to include the option to save the sampled data in a file for later examination. The format of the data file is shown in *sar*(8). In addition to the system counters, a time stamp is also included. It gives the time at which the sample was taken.
- If no frequency arguments are supplied, it generates system activity reports for a specified time interval from an existing data file that was created by *sar* at an earlier time.

A convenient usage is to run *sar* as a background process, saving its samples in a temporary file but sending its standard output to */dev/null*. Then an experiment is conducted after which the system activity is extracted from the temporary file. The *sar*(1) manual entry describes the usage and lists various types of reports. Attachment 11.3 gives formula for deriving each reported item.

### The "sag" command

*Sag* displays system activity data graphically. It relies on the data file produced by a prior run of *sar* after which any column of data or the combination of columns of data of the *sar* report can be plotted. A fairly simple but powerful command syntax allows the specification of cross plots or time plots. Data items are selected using the *sar* column header names. The *sar*(1G) manual entry describes its options and usage. The system activity



graphical program invokes **graphics(1G)** and **tplot(1G)** commands to have the graphical output displayed on any of the terminal types supported by **tplot**.

#### The "timex" command

The **timex** command is an extension of the **time(1)** command. Without options, **timex** behaves exactly like **time**. In addition to giving the time information, it also prints a system activity report derived from the system counters. The manual entry **timex(1)** explains its usage. It should be emphasized that the user and sys times reported in the second and third lines are for the measured process itself including all its children while the remaining data (including the cpu user % and cpu sys %) are for the entire system.

While the normal use of **timex** will probably be to measure a single command, multiple commands can also be timed; either by combining them in an executable file and timing it, or more concisely, by typing:

```
timex sh -c "cmd1; cmd2; ... ;"
```

This establishes the necessary parent-child relationships to correctly extract the user and system times consumed by **cmd1**, **cmd2**, ... (and the shell).

#### The "sadb" command

**Sadb** is a user level program that can be invoked independently by any user. It requires no storage or extra code in the operating system and allows the user to specify the disks to be monitored. The program is reawakened every second, reads system tables from **/dev/kmem**, and extracts the required information. Because of the 1 second sampling, only a small fraction of disk requests are observed; however, comparative studies have shown that the statistical determination of disk locality is adequate when sufficient samples are collected.

In the operating system, there is an **iobuf** for each disk drive. It contains two pointers which are head and tail of the I/O active queue for the device. The actual requests in the queue may be found in three buffer header pools—system buffer headers for block I/O requests, physical buffer headers for physical I/O requests, and swap buffer headers for swap I/O. Each buffer header has a forward pointer which points to the next request in the I/O active queue and a backward pointer which points to the previous request.

**Sadb** snapshots the **iobuf** of the monitored device and the three buffer header pools once every second during the monitoring period. It then traces the requests in the I/O queue, records the disk access location, and seeks distance in buckets of 8 cylinder increments. At the end of monitoring period, it prints out the sampled data. The output of **sadb** can be used to balance load among disk drives and to rearrange the layout of a particular disk pack. The usage of this command is described in manual entry **sadb(1)**.

#### Daily Report Generation

The previous part described the commands available to users to initiate activity observations. It is probably desirable for each installation to routinely monitor and record system activity in a standard way for historical analysis. This part describes the steps that a system administrator may follow to automatically produce a standard daily report of system activity.

#### Facilities

- **sadc**—The executable module of **sadc.c** (see Attachment 11.1) which reads system counters from **/dev/kmem** and records them to a file. In addition to the file argument, two frequency arguments are usually specified to indicate the sampling interval and number of samples to be taken. In case no frequency arguments are given, it writes a dummy record in the file to indicate a system restart.
- **sal**—The shell procedure that invokes **sadc** to write system counters in the daily data file **/usr/adm/sa dd** where **dd** represents the day of the month. It may be invoked with sampling interval and iterations as arguments.



- **sa2**—The shell procedure that invokes the **sar** command to generate daily report `/usr/adm/sa/sar dd` from the daily data file `/usr/adm/sa/sa dd`. It also removes daily data files and report files after 7 days. The starting and ending times and all report options of **sar** are applicable to **sa2**.

#### Suggested Operational Setup

It is suggested that the **cron(1M)** control the normal data collection and report generation operations. For example, the sample entries in `/usr/lib/crontab`:

```
0 * * * 0,6 su sys -c "/usr/lib/sa/sa1 "
0 18- * * 1-5 su sys -c "/usr/lib/sa/sa1 "
0 8-17 * * 1-5 su sys -c "/usr/lib/sa/sa1 1200 3 "
```

would cause the data collection program **sadc** to be invoked every hour on the hour. Moreover, depending on the arguments presented, it writes data to the data file one to three times at every 20 minutes. Therefore, under the control of **cron(1M)**, the data file is written every 20 minutes between 8:00 and 18:00 on weekdays and hourly at other times.

Note that data samples are taken more frequently during prime time on weekdays to make them available for a finer and more detailed graphical display. It is suggested that **sa1** be invoked hourly rather than invoking it once every day; this ensures that if the system crashes data collection will be resumed within an hour after the system is restarted.

Because system activity counters restart from zero when the system is restarted, a special record is written on the data file to reflect this situation. This process is accomplished by invoking **sadc** with no frequency arguments within `/etc/rc` when going to multiuser state:

```
su adm -c "/usr/lib/sa/sadc /usr/adm/sa/sa'date +%d' "
```

**Cron(1M)** also controls the invocation of **sar** to generate the daily report via shell procedure **sa2**. One may choose the time period the daily report is to cover and the groups of system activity to be reported. For instance, if:

```
0 20 * * 1-5 su sys -c "/usr/lib/sa/sa2 -s 8:00 -e 18:00 -i 3600 -uybd "
```

is an entry in `/usr/lib/crontab`, **cron** will execute the **sar** command to generate daily reports from the daily data file at 20:00 on weekdays. The daily report reports the CPU utilization, terminal device activity, buffer usage, and device activity every hour from 8:00 to 18:00.

In case of a shortage of the disk space or for any other reason, these data files and report files can be removed by the superuser. The manual entry **sar(8)** describes the daily report generation procedure.



## ATTACHMENT 11.1

The source files and shell programs of the system activity package are in directory */usr/src/cmd/sa*.

|                            |                                                                                                                                                                                                                                                                                                                                                 |
|----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>sa.h</b>                | The system activity header file defines the structure of data file and device information for measured devices. It is included in <b>sadc.c</b> , <b>sar.c</b> , and <b>timex.c</b> .                                                                                                                                                           |
| <b>sadc.c</b>              | The data collection program that accesses <i>/dev/kmem</i> to read the system activity counters and writes data either on standard output or on a binary data file. It is invoked by the <b>sar</b> command generating a real-time report. It is also invoked indirectly by entries in <i>/usr/lib/crontab</i> to collect system activity data. |
| <b>sar.c</b>               | The report generation program invokes <b>sadc</b> to examine system activity data, generates reports in real-time, and saves the data to a file for later usage. It may also generate system activity reports from an existing data file. It is invoked indirectly by <b>cron</b> to generate daily reports.                                    |
| <b>saghdr.h</b>            | The header file for <b>saga.c</b> and <b>sagb.c</b> . It contains data structures and variables used by <b>saga.c</b> and <b>sagb.c</b> .                                                                                                                                                                                                       |
| <b>saga.c &amp; sagb.c</b> | The graph generation program that first invokes <b>sar</b> to format the data of a data file in a tabular form and then displays the <b>sar</b> data in graphical form.                                                                                                                                                                         |
| <b>sa1.sh</b>              | The shell procedure that invokes <b>sadc</b> to write data file records. It is activated by entries in <i>/usr/lib/crontab</i> .                                                                                                                                                                                                                |
| <b>sa2.sh</b>              | The shell procedure that invokes <b>sar</b> to generate the report. It also removes the daily data files and daily report files after a week. It is activated by an entry in <i>/usr/lib/crontab</i> on weekdays.                                                                                                                               |
| <b>timex.c</b>             | The program that times a command and generates a system activity report.                                                                                                                                                                                                                                                                        |
| <b>sadp.c</b>              | The program that samples and reports disk activities.                                                                                                                                                                                                                                                                                           |



## ATTACHMENT 11.2

```

struct sysinfo {
    time_t      cpu[4];
#define CPU_IDLE      0
#define CPU_USER      1
#define CPU_KERNEL    2
#define CPU_WAIT      3
    time_t      wait[3];
#define W_IO          0
#define W_SWAP        1
#define W_PIO         2
    long        bread;
    long        bwrite;
    long        lread;
    long        lwrite;
    long        phread;
    long        phwrite;
    long        swapin;
    long        swapout;
    long        bswapin;
    long        bswapout;
    long        pswitch;
    long        syscall;
    long        sysread;
    long        syswrite;
    long        sysfork;
    long        sysexec;
    long        runque;
    long        runocc;
    long        swpque;
    long        swpocc;
    long        iget;
    long        namei;
    long        dirblk;
    long        readch;
    long        writech;
    long        rcvint;
    long        xmtint;
    long        mdmint;
    long        rawch;
    long        canch;
    long        outch;
    long        msg;
    long        sema;
};

```



## ATTACHMENT 11.3

The derivation of the reported items is given in this attachment. Each item discussed below is the data difference sampled at two distinct times  $t_2$  and  $t_1$ .

**CPU Utilization**

$$\% \text{-of-cpu-x} = \text{cpu-x} / (\text{cpu-idle} + \text{cpu-user} + \text{cpu-kernel} + \text{cpu-wait}) * 100$$

where cpu-x is cpu-idle, cpu-user, cpu-kernel (cpu-sys), or cpu-wait.

**Cached Hit Ratio**

$$\% \text{-of-cached-I/O} = (\text{logical-I/O} - \text{block-I/O}) / \text{logical-I/O} * 100$$

where cached I/O is cached read or cached write.

**Disk or Tape I/O Activity**

$$\begin{aligned} \% \text{-of-busy} &= \text{I/O-active} / (t_2 - t_1) * 100; \\ \text{avg-queue-length} &= \text{I/O-resp} / \text{I/O-active}; \\ \text{avg-wait} &= (\text{I/O-resp} - \text{I/O-active}) / \text{I/O-ops}; \\ \text{avg-service-time} &= \text{I/O-active} / \text{I/O-ops}. \end{aligned}$$

**Queue Activity**

$$\begin{aligned} \text{avg-x-queue-length} &= \text{x-queue} / \text{x-queue-occupied-time}; \\ \% \text{-of-x-queue-occupied-time} &= \text{x-queue-occupied-time} / (t_2 - t_1); \end{aligned}$$

where x-queue is run queue or swap queue.

**The Rest of System Activity**

$$\text{avg-rate-of-x} = x / (t_2 - t_1)$$

where x is swap in/out, blks swapped in/out, terminal device activities, read/write characters, block read/write, logical read/write, process switch, system calls, read/write, fork/exec, iget, namei, directory blocks read, disk/tape I/O activities, message or semaphore activities.